Power analysis of a hardware accelerated AES implementation

Gustavo Zambonin, Marcello Klingelfus

Operating Systems II (UFSC-INE5424)

1 Motivation

Devices in the Internet of Things are expected to interact with other machinery in diverse situations. As such, threats to these communications should be reduced or nullified altogether by means of protecting the hardware or software from revealing critical information. However, these features may need expensive calculations or precise timing. Hence, reduced energy consumption and increased performance goals need careful consideration.

A *cryptographic accelerator* can be used to help with these constraints, operating in a more efficient way, by means of calculating steps of a cryptographic algorithm on the hardware itself. Yet, using these features requires a deep understanding of the processor used, and may not be available or enabled on a given operating system.

We also point that these implementations can be attacked non-invasively through power analysis, *i.e.* measuring power consumption and checking for patterns in which secret data may be revealed. Ergo, *constant-time* algorithm implementations can be employed to prevent this undesired behaviour. While these are usually very complex to implement, it is still useful to know if current implementations using accelerators are prone to leak important data.

2 Goals

The underlying system-on-chip (SoC) powering EPOSMote III, called CC2538, is derived from the ARM Cortex-M3 blueprint. It features hardware accelerators for AES and RSA ciphers, and the SHA256 cryptographic hash function [T.I13]. Poly1305, an AES-based message authentication code, is used inside EPOS to achieve data authenticity and integrity, as seen in [RF15]. Hence, it is desirable to *optimise the current implementation for the AES cipher*, making use of the cryptographic cores available directly from the processor, as well as *profile its power usage* within a low-footprint, simplified EPOS instance.

Optionally, it is beneficial to apply techniques to harden the code and making it resilient to power analysis, as well as measure how deeply one may control the hardware accelerated cryptographic directives, *i.e.* if there are only "black-boxes" available or separate steps for each cipher, since one can more finely analyse the power consumption for these separate operations. An example of this study can be seen in [SS16].

3 Methodology

To reach our goal, we will first need to study and fully understand the AES cipher in the context of CC2538, *i.e.* how the cryptographic hardware provided by EPOSMote III's SoC works. We will then implement a driver that uses the hardware accelerated features for AES inside EPOS. Afterwards, its correctness will be tested, and its performance measured against the previous implementations. Finally, we will analyse its power consumption by means of reducing the underlying operating system's noise with regards to background tasks, modifying it accordingly (*e.g.* replacing system components by simpler alternatives, such as the thread scheduler).

4 Tasks

1. Elaborate a detailed project plan;
2. Demonstrate the project's viability;
3. Implement a driver based on the cryptographic accelerator for AES on EPOS;
4. Benchmark the previous AES implementation inside EPOS against code using hardware accelerated routines;

{gustavo.zambonin,marcello.klingelfus}@grad.ufsc.br src/

5. Reduce the operating system footprint and measure the power profile for the hardware accelerated implementation;
6. Compare all implementations and display practical results.

5 Deliverables

1. Project plan (this document);
2. Report consisting of a simple demonstration of technological viability;
3. Code that talks to the AES cryptographic accelerator through EPOS;
4. Report with performance comparison between software and hardware AES implementations on EPOS;
5. Report on power analysis, observed through an oscilloscope, for the new driver implementation;
6. Project demonstration.

6 Schedule

Task	**30/04**	**07/05**	**14/05**	**21/05**	**28/05**	**04/06**	**11/06**	**18/06**	**25/06**
Task 1 — Project plan	E1								
Task 2 — Viability	E1								
Task 3 — AES driver	x	x	E2						
Task 4 — Benchmarks			x	x	E3				
Task 5 — Power analysis			x	x	x	x	E4		
Task 6 — Demonstration							x	x	E5

7 E1 — Basic concepts and technological viability

We start by explaining the concepts related to cryptography needed to understand the body of this work. Namely, we talk about cryptographic hash functions, message authentication codes (MACs), the AES cipher, an example of MAC called Poly1305, and power analysis. Afterwards, we show preliminary results on the EPOSMote III platform and EPOS that allow us to demonstrate the technological viability of this project.

7.1 AES — Advanced Encryption Standard

AES is a block cipher that operates with key sizes of 128, 192 and 256 bits on a state matrix of 4×4 words. The words are polynomials of a Galois field with order 256, chosen for its easy computational representation and solid mathematical basis. It is an iterative cipher, featuring repeated applications (rounds) of four basic operations on the state matrix.

Consider the following pseudocode for the encryption algorithm. Denote n as the size of key, $\ell = \frac{n}{32}$, n_r as the number of rounds ($n_r = 10$ if $n = 128$, $n_r = 12$ if $n = 192$, $n_r = 14$ if $n = 256$) and A as the state matrix.

We briefly discuss the routines mentioned above and refer to [DR02] for detailed explanations.

- SubBytes: substitute every element of A for its corresponding value inside a substitution-box (S-box, built from the multiplicative inverse of elements in the Galois field along with an affine transformation), with the intent of concealing the relationship between key and ciphertext;
- ShiftRows: shift every row of A by a fixed amount, with the intent of scattering the values of the state matrix, avoiding the independent encryption of columns;
- MixColumns: multiply every column of A by a specific polynomial and apply modular reduction, achieving further diffusion;
- AddRoundKey: combine A with the round key through an exclusive or operation;
- KeyExpansion: derives a set of words from the key so these can be used in every round, through the use of efficient and non-linear byte operation, preventing localization of patterns that can reveal the master key.

Algorithm 1 AES ciphering process.

```
Require: $m \in \{0,1\}^{128},\ K \in \cup_{n\in\{128,192,256\}}\{0,1\}^n$    ▷ plaintext and key
Ensure: $c \in \{0,1\}^{128}$    ▷ ciphertext
  $A \leftarrow m$
  $\{k_0 \dots k_{(n_r+1)\cdot\ell}\} \leftarrow$ KeyExpansion($K$)
  $A \leftarrow$ AddRoundKey($A, \{k_0, \dots, k_{\ell-1}\}$)
  for $i \leftarrow 1$ até $n_r - 1$ do
    $A \leftarrow$ SubBytes($A$)
    $A \leftarrow$ ShiftRows($A$)
    $A \leftarrow$ MixColumns($A$)
    $A \leftarrow$ AddRoundKey($A, \{k_{i\cdot\ell}, \dots, k_{(i+1)\cdot\ell-1}\}$)
  end for
  $A \leftarrow$ SubBytes($A$)
  $A \leftarrow$ ShiftRows($A$)
  $A \leftarrow$ AddRoundKey($A, \{k_{n_r\cdot\ell}, \dots, k_{(n_r+1)\cdot\ell-1}\}$)
  $c \leftarrow A$
```

AES is an example of symmetric cryptography, *i.e.* it uses a single key for encryption and decryption. Hence, the routines above need to be easily inverted, but only with the master key. We present the pseudocode for the decryption algorithm and discuss the new routines.

Algorithm 2 AES deciphering process.

```
Require: $c \in \{0,1\}^{128},\ K \in \cup_{n\in\{128,192,256\}}\{0,1\}^n$    ▷ ciphertext and key
Ensure: $m \in \{0,1\}^{128}$    ▷ plaintext
  $A \leftarrow c$
  $\{k_0 \dots k_{(n_r+1)\cdot\ell}\} \leftarrow$ KeyExpansion($K$)
  $A \leftarrow$ AddRoundKey($A, \{k_{n_r\cdot\ell}, \dots, k_{(n_r+1)\cdot\ell-1}\}$)
  for $i \leftarrow n_r - 1$ até 1 do
    $A \leftarrow$ InvShiftRows($A$)
    $A \leftarrow$ InvSubBytes($A$)
    $A \leftarrow$ AddRoundKey($A, \{k_{i\cdot\ell}, \dots, k_{(i+1)\cdot\ell-1}\}$)
    $A \leftarrow$ InvMixColumns($A$)
  end for
  $A \leftarrow$ InvShiftRows($A$)
  $A \leftarrow$ InvSubBytes($A$)
  $A \leftarrow$ AddRoundKey($A, \{k_0, \dots, k_{\ell-1}\}$)
  $m \leftarrow A$
```

- InvShiftRows: shift every row of A by a fixed amount, inverting the correspondent diffusion on the encryption algorithm;
- InvSubBytes: substitute every element of A by its corresponding element on the inverse S-box (obtained by applying the inverse of the affine transformation and then taking the multiplicative inverse in the Galois field);
- InvMixColumns: multiply every columns of A by a specific polynomial (the multiplicative inverse of the polynomial for the corresponding step).

7.2 Cryptographic hash function

Consider two sets, X and Y, with words of length m and n respectively, such that $m > n$. A function $H : X \longrightarrow Y$ can be defined as a hash (or compressing) function, since elements of Y are generally called hashes or digests. H is defined to be deterministic and computationally efficient. To consider H as cryptographic, other restrictions are taken into account:

- Preimage resistance, *i.e.* the function cannot be inverted efficiently;
- Second preimage resistance, *i.e.* given $m_0 \in X$, it should not be computationally feasible to find a different $m_1 \in X$ such that their hashes are equal;
- Collision resistance, *i.e.* it should not be computationally feasible to find any two distinct messages in X with equal hashes;
- Presence of the avalanche effect, wherein a single bit change on the input should change approximately half of the bits on the output.

Note that collision resistance implies second preimage resistance. In the former, an attacker may choose any two messages from X, whereas in the latter one message is fixed and cannot be changed. Hence, it is said that second preimage resistance is much stronger as a security requirement.

7.3 Message authentication code

To create a message that has not only integrity, but also authenticity (*i.e.* one can verify if it really came from a specified sender), a MAC construction has to be employed. It authenticates a message by signing it with a symmetric key. Consequently, there must be a way for two (or more) entities to agree on a key. It follows that, if a key is shared between multiple devices (*e.g.* a sensor network), every device that can verify a MAC can also generate other MACs. It consists of three algorithms that can be computed efficiently:

- Key generation, where on input of a security parameter n (usually a large enough integer), a random key is produced from a key space (*e.g.* the set of all binary words of length n);
- Signature generation, where on input of a key and any message, it produces a tag (a short name for an authenticated message);
- Signature verification, where on input of a tag and a key, it returns a Boolean value on whether the tag was created with that key.

Note that this construction should be able to resist forgeries, that is, it should be computationally unfeasible to guess the MAC for messages that were created with an unknown key. We give an example of MAC algorithm in the form of Poly1305-AES and refer to the original paper for detailed explanations [Ber05]. It is sufficient to know that Poly1305 is a polynomial evaluation modulo $2^{130} - 5$, with the following parameters: a secret key with two values (the key for AES and a random integer with special properties), an unique number (nonce) to be encrypted using AES, and the message to be tagged. The underlying security of the algorithm is based on the nonce encryption mechanism. Observe that AES may be replaced at any time by another cipher.

7.4 Power analysis

A cryptographic algorithm is not only required to present a solid mathematical proof, but also implementations that can resist side-channel attacks. Secret information may be retrieved if it can be finely measured and physically exploited. Cache and timing attacks, introduction of faults by placing the device under extreme conditions, acoustic and electromagnetic monitoring etc. are various forms of side-channel attacks. We focus on the concept of power analysis, in which the power consumption may be studied to obtain sensitive data from a given device, and further restrict our scope to simple power analysis, where one can directly measure and visualize the power consumption collected from cryptographic operations.

Consider the example of a RSA cipher, that works with multiplications and squaring modulo prime numbers. These can be distinguished by looking at the measurements through a standard digital oscilloscope. Statistical analysis of the values and removal of background electrical noise may be required to enhance the observation results.

7.5 Results

We now introduce these concepts to the EPOS environment. First, to ensure that a meaningful test bench is used to test the new driver implementation, we created an application out of the `src/component/poly1305_test.cc` file and executed it to ensure the software implementations for the algorithms work correctly, getting positive results and a meaningful execution time. A simple task to prove the viability for the driver implementation is to actually manipulate the cryptographic core. Hence, many definitions in the form of memory addresses for registers and function prototypes had to be migrated to EPOS, from the official driver library provided by Texas Instruments. We created a dependency graph of the files from that library, starting by `cc2538_foundation_firmware_1_0_1_0/driverlib/.../aes_example.c`. We found out that hardly any files are needed, namely (folder names omitted for clarity):

- source files are `aes.c` (AES driver), `cpu.c` (instruction wrappers), `interrupt.c` (driver for the NVIC — Nested Vectored Interrupt Controller) and `sys_ctrl.c` (driver for the system controller);
- header files are, in addition to corresponding files for each source above, `hw_aes.h`, `hw_flash_ctrl.h`, `hw_nvic.h`, `hw_sys_ctrl.h` (addresses and constants for registers that manage the AES driver, flash ROM, NVIC and system controller, respectively), `hw_ints.h` (defines for types of interruptions) and `hw_types.h` (common types and macros for hardware access).

Hence, we moved these files to the EPOS codebase and adapted them accordingly to compile a test application. We present an application that manages to enable the AES core, and discuss the code in detail.

```
#include <utility/aes-hw/aes.h>
#include <utility/aes-hw/hw/hw_sys_ctrl.h>
#include <utility/ostream.h>

using namespace EPOS;
OStream cout;

typedef struct {
  unsigned char ui8AESKey[16];            // stores the Aes Key
  unsigned char ui8AESKeyLocation;        // location in Key RAM
  unsigned char ui8AESBuf[16];            // input buffer
  unsigned char ui8AESExpectedOutput[16]; // expected results
  unsigned char ui8IntEnable;             // set to true to enable interrupts
} tAESExample;

tAESExample sAESexample[] = {
    {{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00},
     0,
     {0x6c, 0x5f, 0x51, 0x74, 0x53, 0x53, 0x77, 0x5a, 0x5a, 0x5f, 0x57, 0x58,
      0x55, 0x53, 0x06, 0x0f},
     {0x83, 0x78, 0x10, 0x60, 0x0e, 0x13, 0x93, 0x9b, 0x27, 0xe0, 0xd7, 0xe4,
      0x58, 0xf0, 0xa9, 0xd1},
     false},
};

int main() {
  // enable AES peripheral
  HWREG(SYS_CTRL_RCGCSEC) = 0x00000010;

  // mimic start of aes.cc@AESLoadKey
  unsigned char *pui8Key = sAESexample[0].ui8AESKey;
  static unsigned int ui32temp[4];
  unsigned char *pui8temp = (unsigned char *)ui32temp;
  unsigned char i;

  // key address needs to be 4 byte aligned
  for (i = 0; i < KEY_BLENGTH; i++) {
    pui8temp[i] = pui8Key[i];
  }

  // configure master control module
  // enable DMA path to the key store module
  HWREG(AES_CTRL_ALG_SEL) = 0x00000001;

  // clear any outstanding events
  HWREG(AES_CTRL_INT_CLR) = 0x00000001;

  // configure key store module (area, size)
  // 128-bit key size
  HWREG(AES_KEY_STORE_SIZE) = 0x00000001;

  // enable keys to write (e.g. Key 0)
  HWREG(AES_KEY_STORE_WRITE_AREA) = 0x00000001;

  // configure DMAC
  // enable DMA channel 0
  HWREG(AES_DMAC_CH0_CTRL) = 0x000000001;

  // base address of the key in ext. memory
  cout << hex << "AES_DMAC_CH0_EXTADDR before writing: "
    << HWREG(AES_DMAC_CH0_EXTADDR) << endl;
  HWREG(AES_DMAC_CH0_EXTADDR) = (unsigned int)pui8temp;
  cout << hex << "AES_DMAC_CH0_EXTADDR after writing: "
    << HWREG(AES_DMAC_CH0_EXTADDR) << endl;

  // total key length in bytes (e.g. 16 for 1 x 128-bit key)
  HWREG(AES_DMAC_CH0_DMALENGTH) = 0x10;

  return 0;
}
```

This implementation follows the directives in [T.I13, Sec. 22.2.5.4.2.1] and can actually write the base key address to the `AES_DMAC_CH0_EXTADDR` register, with value `0x20000050`. The module is enabled by setting the second bit of the `SYS_CTRL_RCGCSEC` register, as per [T.I13, Sec. 7.7.12, pp. 207]. Afterwards, DMA is enabled, and the module can already fetch the key from external memory, finally writing it to the key store (a protected memory region). If the module is disabled, the base key address will not be written at all. Note that `aes_example.c` features both interrupt and polling implementations; the interrupt drivers may not be needed if polling is used exclusively, thus reducing the amount of code needed to port. Additionally, it showcases only ECB (Electronic Code Book) mode, whereas other modes can be used, according to [T.I13, Sec. 22.2.1.1.3].

Finally, we briefly discuss how to measure the power consumption through an oscilloscope. Preliminary discussion showed that an *amplifier circuit* is needed. One needs to mangle a USB-microUSB cable, acting as power for the EPOSMote III, to insert a shunt resistor and actually read tension from the current passing through the cable. Then, since the signal is too weak to be captured by the scope, one will need an operational amplifier, some resistors (according to this description) and a protoboard to implement an amplifier circuit. Finally, when the scope is able to sense it, the measurements can be taken (this circuit design is subject to changes). One can also disable certain parts of the SoC by making use of the deep sleep feature [T.I13, Sec. 3.5.1, pp. 132], and further simplify the application's traits file, to reduce background noise.

Referências

[Ber05] D. J. Bernstein. The Poly1305-AES Message-Authentication Code. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49, February 2005.

[DR02] J. Daemen and V. Rijmen. *The Design of Rijndael*. 1st edition, 2002.

[RF15] D. Resner and A. A. Fröhlich. Key Establishment and Trustful Communication for the Internet of Things. In *Proceedings of the Fourth International Conference on Sensor Networks*, pages 197–206, February 2015.

[SS16] P. Schwabe and K. Stoffelen. All the AES You Need on Cortex-M3 and M4. In R. Avanzi and H. Heys, editors, *Selected Areas in Cryptography — SAC 2016*, volume 10532 of *Lecture Notes in Computer Science*, pages 180–194, August 2016.

[T.I13] T.I. CC2538 System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee®/ZigBee IP® Applications. User's Guide SWRU319C, Texas Instruments, May 2013.