Performance of Authenticated Encryption for Payment Cards with Crypto Co-processors

Keith Mayes

Royal Holloway, University of London
Egham, Surrey, UK
Email: keith.mayes@rhul.ac.uk

***Abstract*—Many security protocols rely on authentication of communicating entities and encryption of exchanged data. Traditionally, authentication and encryption have been separate processes, however there are combined solutions, referred to as authenticated-encryption (AE). The payment card industry is revising its protocol specifications and considering AE, however there has been uncertainty around performance and feasibility on traditional issued smart cards and when loaded as applications on security chips pre-installed within devices. It is difficult to predict performance using results from generic CPUs as typical smart card chips used in payment, have slow CPUs yet fast crypto-coprocessors. This report is based on a practical investigation, commissioned by a standards body, that compared secure platform level (MULTOS) and low-level native implementations of AE on crypto-coprocessor smart cards. The work also suggests a technology independent benchmark for a CPU with crypto-coprocessor.**

Keywords–Authenticated encryption; EMV; OCB; GCM; ETM; CCM; smart card; crypto-coprocessor; payment; performance; MULTOS.

I. INTRODUCTION

The EMVCo organisation [4] developed the Europay, Mastercard and Visa (EMV) standards [3] that affect billions of payment smart cards. The cards use secured microcontroller chips, designed to be strongly tamper-resistant and independently evaluated to Common Criteria (CC) [2] levels of at least Evaluation Assurance Level (EAL)4+. Despite strong defensive capabilities, the chips lag behind the state-of-the-art in CPU performance and memory sizes. However, despite these limitations the chips excel in cryptographic operations as they incorporate relatively high-speed crypto-coprocessor hardware. The EMVCo organisation is reviewing the use of Authenticated Encryption (AE) [10] for future payment card processing. There are a number of potential modes and those originally of interest included Offset Codebook (OCB) [15], Galois Counter Mode (GCM) [20], Counter with Cipher Block Chaining Message Authentication Code (CCM) [19] and Encrypt-then-MAC (ETM) [10]. Within this study, GCM was eventually substituted for OCB3 as the former required binary field multiplication, which was not supported by the available crypto-coprocessors. There have been previous studies of AE performance, however they have generally focussed on more powerful generic CPUs, without dedicated crypto-coprocessors. As a starting point we take the study by Krovetz and Rogaway [14], which shows that OCB performance is faster (for the given test conditions) than alternatives; however there are several reasons why these results cannot be immediately accepted as relevant for EMV protocols:

- The command messages in traditional smart cards are small; the data field restricted to 255 bytes; larger payloads accommodated by multiple messages.
- The results do not adequately address the case of a slow CPU with a relatively fast crypto-coprocessor.
- Support for Associated Data is not required.
- Smart cards have very restricted memory sizes with different write speeds for Random Access Memory (RAM) and Non-Volatile Memory (NVM).
- Conventional smart card interfaces are quite slow and so protocols can be communication limited rather than processing limited.

In order to gain a better appreciation of the comparative performance of AE on realistic smart card platforms, a practical study was initiated, considering first a secure platform implementation (MULTOS) [17] and then a native mode equivalent. This report describes the experimental requirements in Section II and then gives an overview of the AE modes in Section III. The platform and native results are presented and discussed in Sections IV and V respectively. Section VI discusses how implementation security may affect performance measurements, and Section VII considers communication limitations. Conclusions and suggestions for future work are presented in Section VIII.

II. EXPERIMENTAL REQUIREMENTS

The study investigated comparative performance of AE modes implemented in both a secured smart card application platform (representative of a pre-deployed device), and as native code on a smart card chip. The selected platform was a MULTOS ML3 card, using the Infineon SLE78 chip [7], which can be CC EAL4+ certified, and includes good defences against physical, side-channel [12][13] and fault attacks. The native mode implementation used a Samsung 16-bit smart card chip (S3CC9E8) [23], and as the crypto-coprocessor did not support AES, its performance comparisons used 3DES/DES [5]. The S3CC9E8 is a secured microcontroller with physical attack protection, fault sensors and some side-channel countermeasures, however it would normally require added defensive measures in software; this is discussed further in Section VI. The AE modes considered in detail were OCB (OCB2 and OCB3), CCM and ETM; with some GCM experiments.

 ISBN: 978-1-61208-547-0

The EMV protocol would normally have a preliminary Diffie Hellman key and nonce exchange, however this was not modelled as would be common to all AE modes and so would not affect performance comparison. Associated Data is not needed in the EMV protocol. Communicated data is required to fit within one or more standard Application Data Protocol Units (APDU) [8], and with the exception of OCB modes, all APDU payloads that are not multiples of the encryption block-size are padded prior to encryption. The memory in smart card chips is very restricted and protocol/algorithm execution is expected to place very limited demands on it, leaving maximum space for OS and applications. For our tests, a working assumption was that 80-90% of the memory was unavailable. The RAM in smart cards is usually much faster for writing than the NVM and so critical objects/buffers are implemented in a RAM. Our application was limited to no more than 10% of the available RAM (so if 8k, we could have 800 bytes). The application was restricted to no more than 10% of the available code/data space (so if a 64k flash device then 6.4kbytes was allowed). Some implementations benefit from trading NVM space for speed using pre-computed tables, which is not well suited to smart cards, but up to 10% of the NVM space was assumed available for this. In general the imposed memory restrictions proved not to be a problem for the implemented AE modes.

Test software was in 'C', so it could be adapted and directly comparable for both MULTOS and native implementations. There is a single test application that incorporates all the AE modes plus test utilities that measure various core functions. The interface is based on APDU commands and responses, with the payload data consisting of blocks of plaintext or ciphertext. For message timing precision, commands were run 1024 times before response, in order to compensate for measurement tolerance. Communication delay was removed (via calibration) from the test results, although it is reconsidered in Section VII. We will now continue the discussion by providing an overview of the AE modes.

III. Overview of Authenticated Encryption Modes

Offset Codebook mode is defined as mechanism 1 in ISO/IEC 19772 [10] and is also described in RFC 7253[15]. The principles of operation are also well presented on Phil Rogaway's website [21]. For convenience, we will summarise the basic operations of OCB2 here. In Figure 1, an initialisation vector is first computed and then the plaintext message is split into blocks (M1-3, M* in example), all but the last block must be the size of the block cipher, so for AES128 we have 128 bit blocks. They are then encrypted (with modification from the input vector) to produce ciphertext blocks. The complete output is the sequence of C1-3, C* plus an extra value T. Note that because of a requirement to recompute the intialisation vector, this AE is most optimum for a 64 block message sequence and least optimum for a single block message.

CCM is mechanism 3 in ISO/IEC19772[10] and described in NIST SP800-38C[19] and [24]. Figure 2 overviews CCM operation. Whilst the simplified diagram just shows a nonce/-counter input to the stages of the MAC calculation, the generic standard description also specifies some flag/length bit fields.

ETM scheme (see Figure 3) is mechanism 5 in ISO/IEC 19772 [10], and is a conventional approach with separate

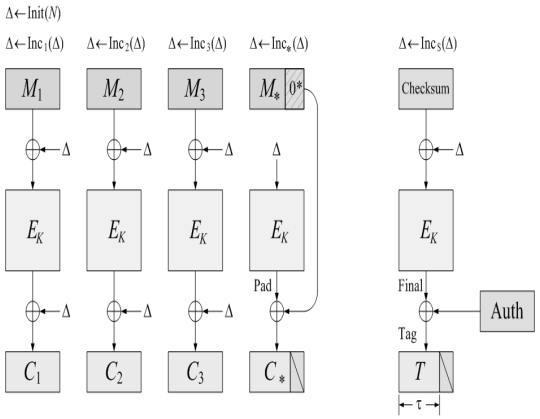

Figure 1. OCB with Incomplete Blocks [Rogaway]

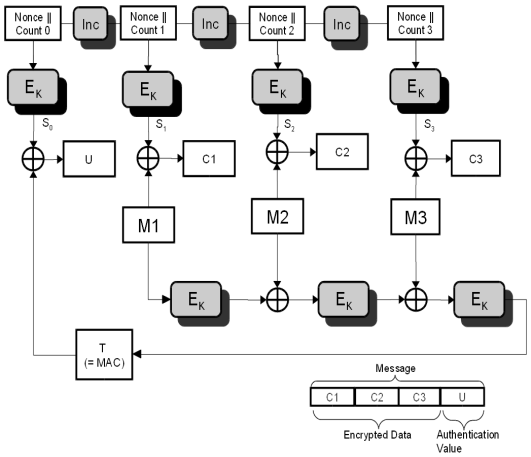

Figure 2. CCM Overview (simplified)

encryption and MAC processes. It does not support Associated Data, although this is not required for the study. The encryption stage uses block encryption in counter mode with key K, followed by a MAC computation on the cipher text using a different key (K') to that used for encryption. According to ISO/IEC 19772[10] the MAC algorithm is selected from the ISO/IEC 9797 standards [11], in which there are six different MAC options, all of which have numerous variants. The selected options for the tests are listed below.

- MAC Algorithm: 1 (usually referred to as CBC-MAC)
- Padding Method: 1 (zeros)
- Final Iteration: 1 (same as other iterations)
- Output Transformation: 1 (unity = no change)
- Truncation: - (left most 64 bits)

GCM (see Figure 4) mode of operation is mechanism 6 in ISO/IEC 19772 [10] and also described in NIST SP800-38D [20] and [22]. The performance of this mode could not be very usefully compared using the traditional crypto-coprocessors used for the study as GCM requires support for multiplication over Galois Field GF (2^{128}) with the hash key H, which is the encryption of all zeros under E_K.

A. Workload Estimation

Table I gives an indication of the underlying workload for each mode when processing the representative test message sizes (as advised by the commissioning standards body).

 ISBN: 978-1-61208-547-0

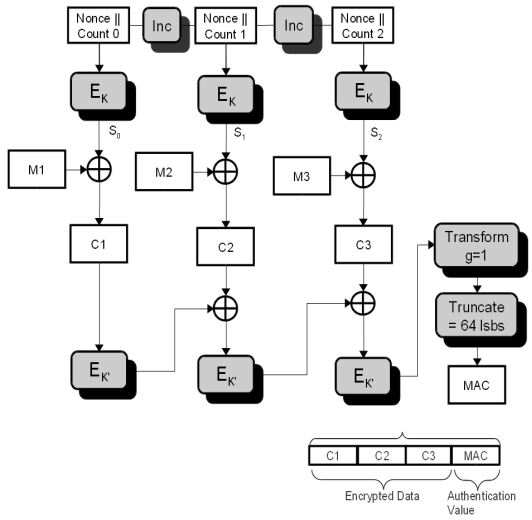

Figure 3. Encrypt then MAC

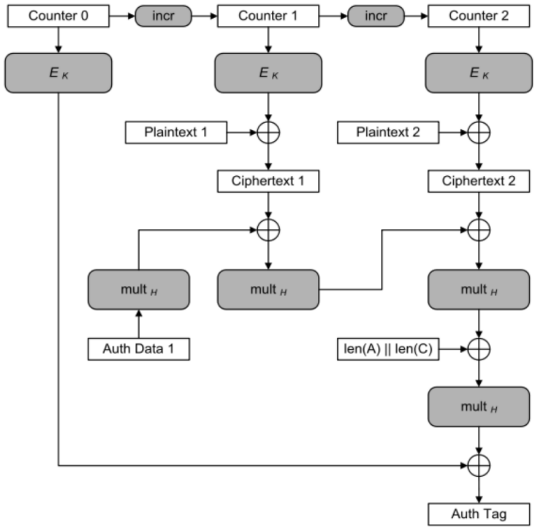

Figure 4. CCM Overview (simplified)

IV. Platform Mode Results

For security, certification and reliability reasons, it is not normal to have native code access to a smart card or similar security chip once deployed. Instead the chip may offer a secure platform where added functionality is constrained to a tightly controlled application layer, using APIs to access security capabilities. The MULTOS card is such a secure platform whereby the application execution language is abstracted from the underlying hardware (see [18]), offering high standards of security, but making it difficult to predict performance of the core AE functionality. The results of initial benchmark tests

TABLE I. ALGORITHM WORKLOAD PER MODE

Bytes	Blks	Msgs	OCB		GCM		CCM	ETM
			E	Init	E	Mul	E	E
8	1	1	3	1	2	2	3	2
16	1	1	3	1	2	2	3	2
20	2	1	4	1	3	3	5	4
32	2	1	4	1	3	3	5	4
40	3	1	5	1	4	4	7	6
64	4	1	6	1	5	5	9	8
128	8	1	10	1	9	9	17	16
192	12	1	14	1	13	13	25	24

TABLE II. MULTOS BENCHMARK MEASUREMENTS (ms)

Function	Primitive		Application		Used
	RAM	NVM	RAM	NVM	
Block Encrypt	3.3	6.4			3.3
Block Xor	0.73	3.94	3.21	15.84	0.73
Block Shift	1.24		2.7		1.24
Block Copy	0.36		0.65		0.36
GF Multiply			199		199

are shown in Table II.

The time measured for a block encrypt with a 128-bit key was 3.3ms (confirmed by MULTOS as matching in-house results). The underlying chip crypto-engine is much faster, and the speed disparity is due to software reliability and security measures. The 3.3ms is only valid when writing encrypted data to RAM, as NVM increases the time to 6.4ms (although reading from NVM is fast); so the outputs of all functions were written to RAM. In all cases where a primitive was available, it was considerably quicker than any equivalent implemented at the application layer, although considerably slower than what might be imagined from a low-level native implementation

GCM requires a finite field multiply, but such a function did not exist as a MULTOS primitive and so was provided in a simple implementation similar to *Algorithm 1* in the standard [16]. Multiplying a single block takes 199ms, even when using primitives *multosBlockShiftRight* and *MultosBlockXor*. Other implementations are described in the standard, although they make use of time/memory trade-offs, which is not a strength for a memory limited smart card. For the initial tests, all the modes and the extra test utilities were built into a single application with the following memory requirements.

- Code Size (NVM): 5701 bytes
- Static Data (NVM): 498 bytes
- Session Data (RAM): 113 bytes

All the sizes are well within the realistic and practical design targets defined at the start of the project. For a single mode application the code size would be considerably less, and the static data is mainly internally stored test-vectors that would not normally be present. The session data could be reduced, if required.

A. Initial Tests and Optimisation

Following the MULTOS benchmark tests, the GCM mode was removed from the study (on request of the commissioning standards body) and more attention given to OCB (version 2) optimisation; and later OCB3 was also added. GCM requires specialist hardware support that was not available from the crypto-coprocessors in the test chips, whereas the other AE modes could be implemented in a straightforward manner. OCB2 was initially implemented from the published example code (see Figure 5)that was critically dependent on a function called *two_times()*.

This was replaced with a version (with less shifts) more suited to the MULTOS Platform (see Figure 6), which had a marked improvement on performance.

Given the resulting speed-up (four/five times on larger messages) from improving OCB2 code, it was decided to also implement OCB3 based on the pseudo code and test vectors in RFC7253 [14].

 ISBN: 978-1-61208-547-0

```
//128-bit shift-left src <<= 1, XOR 0x87 if carry out
 {unsigned i;
 unsigned char carry=src[0]>>7;
// carry = high bit of src
 for (i=0; i<sizeof(block)-1; i++) {
  dst[i]=(src[i]<<1)|(src[i+1]>>7); }
  dst[sizeof(block)-1]=(src[sizeof(block)-1]<<1)
                       ^(carry*0x87);
  }
```

Figure 5. Published Example Code for *two_times()*

```
static void two_times(block dst, block src)
{
 unsigned char carry = src[0] & 0x80;
 multosBlockShiftLeft(AES_BLK_SZ, 1, src, src);
 if (carry) {src[AES_BLK_SZ - 1] ^=  0x87;}
 }
```

Figure 6. MULTOS Code for *two_times()*

1) OCB3 Memory considerations: At the beginning of the OCB3 encrypt pseudo code, a number of bit arrays need to be set-up, see Figure 7, noting that '_' is used to indicate subscript in the pseudo code and that *double()* is the same as the *two_times()* function used in OCB2. The array L_i to use in block processing, varies per message block using index L_[*ntz(i)*]. **L_i:** If we allow for processing 64 blocks of 128

```
L_* = ENCIPHER(K, zeros(128))
L_$ = double(L_*)
L_0 = double(L_$)
L_i = double(L_{i-1}) for every integer i > 0
```

Figure 7. OCB3 Key-dependent Variable Set-up

bits then it might appear that we need 64 of the L_i arrays. However the *ntz(i)* index means we only need 6 (2^6 = 64) L_i arrays, as well as L_*, L_$ and L_0. Therefore we need 9 blocks (144 bytes), rather than 67 blocks; which is well within our target RAM limit.

ntz(): Another memory requirement arises from the ntz() function. Bit/byte manipulations at the MULTOS application layer are slow and so it is quicker to implement the function as a look up table. For a maximum 64 block message we require a 64 byte array that can be precomputed and stored in NVM. This small amount of memory is easily accommodated within a smart card.

2) OCB3 Functional Aspects: OCB3 defines a hash function for use with Associated Data, however this is not needed in the EMV experiments. OCB3 has a preparation stage where key and nonce related data is readied prior to processing message blocks. The key data was described earlier (computation is relatively straight forward) and nonce related data is illustrated in Figure 8. This is mostly straightforward apart from the innocuous looking line showing the calculation of *Offset_0*. The variable *bottom* will have a value between 0 and 63; and it is effectively used as a bit-wise left shift. As discovered

```
Nonce = num2str(TAGLEN mod 128,7)
        || zeros(120-bitlen(N))||1||N
bottom = str2num(Nonce[123..128])
Ktop = ENCIPHER(K, Nonce[1..122]||zeros(6))
Stretch = Ktop||(Ktop[1..64] xor Ktop[9..72])
Offset_0 = Stretch[1+bottom..128+bottom]
Checksum_0 = zeros(128)
```

Figure 8. OCB3 Nonce and Pre-encrypt Variables

TABLE III. MULTOS PLATFORM RESULTS (ms)

Bytes	OCB2	CCM	ETM	OCB3
8	16.59	17.78	14.27	28.66
16	16.61	17.22	13.70	29.27
20	22.17	25.73	22.21	34.40
32	22.17	25.16	21.62	35.00
40	27.72	33.67	30.15	40.12
64	33.35	41.09	37.57	46.42
128	55.77	72.91	69.38	69.21
192	78.17	104.73	101.22	92.06

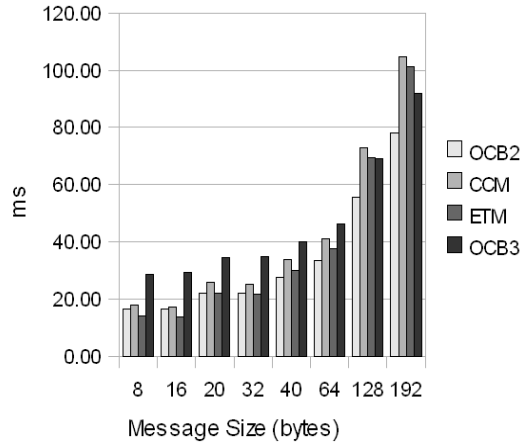

Figure 9. AE Comparative Performance on MULTOS Platform

previously, application level bit-shifts are inefficient on the MULTOS test platform, however the primitives *multosBlock-ShiftLeft/Right* are much quicker. Unfortunately, the primitives require a fixed constant value for the number of places to shift. Although the operation is only carried out once per message it could adversely affect efficiency, especially of small messages and so effort was directed towards optimisation. The first step was to split *bottom* into a number of byte shifts plus a smaller number (up to seven) bit shifts. Byte shifts are easy as we can just change the array index. The bit-shifts were used in a switch/case to reach primitive calls with the appropriate number of shifts. More code was needed, but the overall code space requirements are small.

B. MULTOS Platform Results

The results from testing OCB2, CCM, ETM and OCB3 are shown in Table III.

From the MULTOS results we can see OCB2 is the quickest mode for message sizes beyond 32bytes. OCB3's initial processing makes it slower than OCB2, and OCB3 only overtakes ETM for messages larger than 128 bytes. CCM is always a little slower than ETM due to the extra encryption block, and both are less efficient when working on input data that requires padding.

Although OCB2 seems the faster option for the MULTOS platform (for messages 32+bytes) the relative difference in processing time is not enormous. OCB2 benefited from some optimisation, however there is little scope for improvement in ETM and CCM as much of their time is spent encrypting, which is only possible via a MULTOS API call. The MULTOS platform (and platforms in general) add abstraction between the application layer and the underlying hardware, and so there is considerable uncertainty that the comparative results of Table III would be similar in a native mode smart card implementation. Furthermore, the absolute performance

 ISBN: 978-1-61208-547-0

TABLE IV. TDES MASKED MODE AE TIMES (ms)

Bytes	OCB2	CCM	ETM	OCB3
8	3.04	2.16	1.53	5.75
16	3.07	2.12	1.49	5.81
20	4.19	3.48	2.85	6.73
32	4.24	3.43	2.80	6.81
40	5.37	4.77	4.15	7.76
64	6.57	6.04	5.42	8.81
128	11.23	11.28	10.65	12.82
192	15.89	16.51	15.89	16.82

times on the MULTOS platform, would be expected to be at least one order of magnitude slower than a simple native implementation. Therefore, the AE modes were next tested on a hardware emulator for an older, but still relevant 16-bit smart card chip (Samsung S3CC9E8).

V. NATIVE MODE

Obtaining a native mode hardware emulator for a ”real” smart card with crypto-coprocessor (for use in academic research) is not trivial and only the S3CC9E8 emulator/chip was suitable and used in payment cards; although because it did not support AES, substitute 16 byte block encryption functions were needed. To ensure that comparative performance results would be relevant to standards, the commissioning standards body was consulted on the substitutes. The AES 16byte data block was considered as a pair of 8byte data blocks (M1 and M2) to be coded with DES or triple DES (TDES), i.e., TDES(M1)||(M2) or DES(M1)||(M2). Clearly these functions were for performance evaluation only, although TDES(M1)||TDES(M2) was also coded as a more secure, but overly co-processor intensive alternative.

A. Initial Implementation and Measurement

This stage was focussed on porting the MULTOS code to the native emulator and generating early raw results for functional checking. They derive from non-optimised code, simply replacing the MULTOS primitive calls with equivalents. The performance of the AE modes (including OCB3) was measured in a similar way to the MULTOS work. The first tests used the dual TDES(M1)||TDES(M2) block encryption option (hardest to compute) and the results are in Table IV.

From these initial native results, we observe that the processing time for a single message was under 17ms, regardless of the AE mode. Although the block ciphers were of course different, the overall native execution times were significantly faster than those from the MULTOS experiments, even without optimisation. ETM was the best option for single APDU messages, although in absolute terms there was not much to choose between any of the modes. For smaller messages, ETM and CCM still seemed to have the advantage over the OCB modes. Common to both native and MULTOS implementations ETM is always a little better than CCM and OCB3 does not seem to improve on OCB2.

B. Optimisations

The original source code used within the initial tests was very similar to the MULTOS code. The scope for optimisation on the MULTOS platform was limited as core functions were most efficiently carried out using platform primitives that were abstracted from the underlying hardware. Native mode programming generally offers more opportunity for optimisation as there is less hardware abstraction. Only speed optimisation was considered in this part of the study as all versions of the native code were well within our target memory bounds.

TABLE V. OPTIMISATION OF CORE FUNCTION EXECUTION (ms)

Function	Original	Optimised
Block Xor	0.161	0.071
Block Copy	0.114	0.064
ECB TDES || TDES + mask	0.608	0.381
Fixed Block Shift Left	0.330	0.073

Data Block Copy and XOR: The algorithm modes make use of simple byte manipulation functions including XOR and Copy. In the MULTOS implementation these functions were provided by MULTOS primitives, which in the native code were initially replaced by simple equivalents that assumed variable sized fields and handled data byte-by-byte. However, within the authentication modes, very few operations use variable sized fields, with the majority working on 16 byte memory blocks. Knowing the field size, means that we can avoid loop counters, and by ensuring that the blocks are aligned on 4-byte boundaries we can perform operations on unsigned long integer types rather than bytes. Referring to Table V we see that as a result, BlockXor and BlockCopy have almost doubled in speed, which has also improved the overall block cipher performance. Note that functional calls are still used at this stage rather than in-line code.

Block Shifts: The OCB modes use Copy and XOR operations, but also rely on the function *two_times()* (discussed earlier), which in turn makes use of a function for shifting the contents of a block to the left. The function from the first tests, *BlockShiftLeft()* was a direct replacement for the MULTOS primitive that supported variable shifts on variable sized blocks, referred to by pointer parameters. However, in practice, *two_times()* can be constrained to always use shifts of one place in a 16 byte global variable block. It was therefore possible to create a simpler *FixBlockShiftLeft()* function to use instead. The resulting speed improvement for the shift functions was very significant, as shown in Table V.

Further Refinement: When implementing the block cipher functions, further optimisation removed calls to core functions involving variable length arguments, and in some cases replaced them with simple in-line code. The block encryption function no longer called the core functions, but had faster in-line equivalents. The different block functions are handled by compile-time switches. Note that when using a crypto-coprocessor an input may be masked to reduce side-channel leakage and so a dummy mask was included in the test modes. An option was also added to clear the keys after use, however this was not used in the main measurements. The extended set of benchmarked measurements is shown in Table VI, however now that operations are speed optimised the absolute figures are significantly influenced by the measurement command handling. It is more useful to consider the relative measurements, e.g., by subtracting the FixBlockCopy time from the others.

C. Native Mode Results

Following the additional optimisations, the message tests were repeated for the substitute block cipher function TDES(M1)||M2. The functions are clearly intended to assess performance, rather than to ensure security of the data. The results are provided in Table VII and shown graphically in Figure 10.

 ISBN: 978-1-61208-547-0

TABLE VI. OPTIMISED CORE PERFORMANCE BENCHMARKS (ms)

Functionality	Time
FixBlockXor	0.071
FixBlockCopy	0.064
FixBlockShiftLeft	0.073
DES(M1)||M2	0.128
DES(M1)||DES(M2)	0.141
DES(M1)||DES(M2) + mask XOR	0.146
DES(M1)||DES(M2) + mask XOR + key clear	0.154
TDES(M1)||M2	0.140
TDES(M1)||TDES(M2)	0.163
TDES(M1)||TDES(M2) + mask XOR	0.169
TDES(M1)||TDES(M2) + mask XOR + key clear	0.178

TABLE VII. TDES(M1)||M2 AE TIMES (ms)

Bytes	OCB2	CCM	ETM	OCB3
8	0.54	0.34	0.27	0.83
16	0.57	0.30	0.23	0.79
20	0.65	0.50	0.43	0.92
32	0.70	0.45	0.38	0.91
40	0.79	0.64	0.57	1.07
64	0.95	0.75	0.68	1.16
128	1.46	1.35	1.28	1.65
192	1.96	1.95	1.88	2.14

D. Observations on the Native Tests

Considering Table VI we have significantly improved the performance of core functions. We can also use these results to estimate the achievable raw speed of the crypto-coprocessor, by cancelling out the software manipulations. For both DES and TDES operations we set-up the same keys (two are redundant for DES, but help our timing comparison), wrote in the input data once and read out the result once. The DES crypto-engine overwrites its input data with its output and so for TDES the CPU does not need to move data between the sequence of DES executions; it just refers to a different pre-stored key for each execution. Therefore, if we look at the times for an equivalent DES and TDES operation the difference should be the time taken for the extra DES executions. This time is largely dependent on the hardware although the execution has to be started and checked for completion by the CPU. We can estimate the core DES run time t_d using the following example, where $t(f)$ is the time to execute function f.

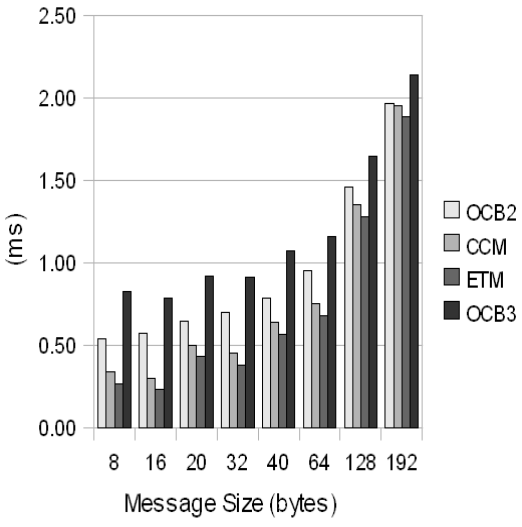

Figure 10. Optimised TDES(M1)||M2 AE Times (ms)

$$\begin{aligned} 2t_d &= t(TDES(M1)||M2) - t(DES(M1)||M2) \\ &= 0.140 - 0.128 \\ &= 0.012ms \end{aligned} \quad (1)$$

There were two extra DES runs in the TDES version so we might suppose that each was about 6us. We can check this by calculating the following.

$$\begin{aligned} 4t_d &= t(TDES(M1)||TDES(M2)) \\ &\quad - t(DES(M1)||DES(M2)) \\ &= 0.163 - 0.141 \\ &= 0.022ms \end{aligned} \quad (2)$$

The four extra DES runs take 22us, about 5.5us each; which is close to our earlier estimate. We can also see from Table VI that the dummy XOR on a 16byte block using in-line code takes about the same time, 5-6us. The key-clear, which is a 24 byte write, takes about 8-9us, so a 16byte block copy should be in a similar 5-6us range. The optimisations improved the speed of all AE modes.

E. Technology Independent Gain Assessment

Generally the native mode results demonstrated that for the particular chip, the crypto-coprocessor could execute its main block cipher in about the same time as the simplest of CPU functions (XOR) on a similar sized block. This could be defined as say the Technology Independent Gain Assessment (TIGA) for any CPU with a crypto-coprocessor. It could be expressed as the percentage of the block encryption that can be completed by the crypto-coprocessor in the time it would take the CPU to compute a block XOR; in our native case this would be 100% and 33% respectively for DES and TDES. In the case of a platform, the benchmark would be computed from the API measurements as we are restricted to the application level. Referring back to the MULTOS measurements in Table II then the TIGA benchmark figure would be approximately 22%. Although we are not comparing like-with-like block ciphers due to practical experimental restrictions, TIGA is at least a means to make comparison. A high figure would suggest that a designer could use block encryptions as readily as XORs and so algorithm optimisation and performance would be quite different to conventional (non crypro-coprocessor) CPUs.

At this point it should be recalled that cards/chips of interest are security sensitive and likely to be attacked. Fortunately countermeasures are quite well understood by the card industry, but they can potentially impact on performance, and so in the next section we consider how our results might be affected.

VI. Implementation Security and Performance

Payment cards safeguard financial transactions of significant value and so are required to strongly resist a wide range of attacks. EMV cards rely on the protection of various stored assets including cryptographic keys, account details and PINs, as well as on the integrity of critical functionality Adhering to information security best practice guidelines for design, (e.g., for algorithms, keys and random number generation) is not at all sufficient as many of the attacks target the implementation rather than the design. In smart cards, the attack resistance will be provided by a mix of hardware and software measures and

 ISBN: 978-1-61208-547-0

so there is potential for performance impact. We can consider such attacks under the following three categories.

- Physical
- Fault
- Side-Channel

A. Physical Attack Resistance

Physical attack generally requires considerable expertise, equipment and time. It may for example involve decapsulating a chip, hardware reverse engineering, probing buses and memories and modifying tracks. However smart card chips have numerous defences against such intrusions, including:

- Passive and active shields - to prevent access to a working chip
- Encrypted buses and memories - to impede direct probing
- Light sensors - to detect decapsulation
- Scrambled circuit layout - to make hardware reverse engineering difficult

Both the chips used in this study incorporate these protective measures, and because they are inherent in the hardware we do not need to degrade our performance test results.

B. Fault Attack Resistance

Fault attacks are active, in that they use means to disrupt the normal operation of the target device (chip); but without damaging it. The faults can, for example, be generated from voltage glitches, radiation pulses and operating the target outside of its operational specification. Under fault conditions the chip may reveal all kinds of information that it would not do when working normally and there are some very elegant attacks including extraction of RSA keys [1]. The hardware sensors in traditional tamper-resistant smart cards (like the S3CC9E8) are intended to detect the likely means of fault insertion and prevent a response useful to the attacker; so there may be no significant added overhead for the software. A sophisticated attack might possibly bypass the sensors, however by adopting openly peer-reviewed algorithms and using diversified card keys, we remove motivation for such effort. Added countermeasures could be to verify a result or to run an algorithm twice and only output a response if the result is valid/consistent, however both strategies rely on the correct outcomes of flag tests and loop counts. It is therefore good practice to add defensive coding of loop and flag tests, at the cost of some additional processing overhead,

The SLE78 chip works very differently to a traditional smart card chip as it has two CPUs working in tandem and a fault is detected if their processing does not agree. This is an innovative and effective approach, which would make it very difficult to succeed with a fault attack. As the protection is inherent in the chip hardware it should not noticeably impact our test results.

C. Side-Channel Attack Resistance

Side-channel leakage implies the leakage of sensitive information (especially keys) via an unintentional channel. This can take the form of key/data-dependent timing variations, power supply fluctuations or electromagnetic emissions. Analysis techniques are well known (see [12] [13]) and can be very powerful against unprotected implementations, including best-practice algorithm designs such as AES. Fortunately, modern smart cards are well protected against such attacks, with a range of countermeasures that mainly impede statistical averaging of signals (used to detect signals in noise) or reduce the source generation of the leakage. Attack countermeasures include:

- Power smoothing
- Noise insertion
- Randomisation of execution
- Timing equalisation
- Dual-rail logic (or Dual CPUs)

The SLE78 chip used in the MULTOS card has a sophisticated dual processing arrangement known as “Integrity Guard” that is believed to be effective at suppressing leakage at source, and this coupled with the Common Criteria certified MULTOS secured OS would suggest that no significant further performance degradation would be incurred from application level countermeasures.

The S3CC9E8 used in the native implementation is a traditional secured microcontroller chip with a single CPU and so it will include some noise smoothing and execution randomisation, but will not suppress the leakage signals at source. Given the age of the chip one would expect some extra side-channel leakage protection to be required from the software, which will have a performance impact. Our tests already included a dummy XOR to represent masking the data used in the crypto-coprocessor, however for this type of chip more help would be needed. One technique used for fast, but perhaps “leaky” crypro-processors is to run the algorithm multiple times, so that an attacker does not know which run used the correct data rather than a dummy pattern. Clearly if you hide your data in a 10 algorithm sequence, you would expect to lose an order of magnitude in performance. Hamming weight equalisation is another technique (used in non-secured CPUs) that seeks to reduce information leakage by ensuring that for each bit transition there is a complementary transition; so as a ‘1’ changes to ‘0’ there is also a ‘0’ changing to ‘1’. In principle this should reduce leakage, however due to electrical, timing and physical layout factors, register bits do not contribute equally to leakage, so the reduction is inferior to hardware measures and may not justify the effort. In a practical implementation this could for example be a 16-bit processor where the lower 8-bits of a register handle the normal data and the upper 8-bits handle the complementary data. This alone is not sufficient as it is necessary to also clear the registers before and after use and so rather than a two-fold reduction in performance, at least an order of magnitude should be anticipated.

D. Observations

It is likely that physical and fault attack protection can be handled by the smart card hardware without significantly degrading performance. For the MULTOS card based on the SLE78 we have sophisticated hardware coupled to an OS designed for the highest levels of security, and Common Criteria evaluation checks for strong protection against side-channel leakage. For the native implementation in the S3CC9E8 we

 ISBN: 978-1-61208-547-0

TABLE VIII. CARD INTERFACE TRANSMISSION TIMES (ms)

Bytes	Contact (bits/s)			Contactless (bits/s)	
	13441	78125	312500	106000	424000
8	4.76	0.82	0.20	0.60	0.15
16	9.52	1.64	0.41	1.21	0.30
20	11.90	2.05	0.51	1.51	0.38
32	19.05	3.28	0.82	2.42	0.60
40	23.81	4.10	1.02	3.02	0.75
64	28.09	6.55	1.64	4.83	1.21
128	76.19	13.11	3.28	9.66	2.42
192	114.28	19.66	4.92	14.49	3.62

would anticipate additional side-channel countermeasures in software and if we consider the techniques in the earlier section then losing an order of magnitude in performance should be expected.

The motivation for a side-channel attack just to capture the EMV session keys is questionable, however discovery of the keys might expose other assets or assist with sophisticated attack strategies. Therefore, it would be prudent to consider an order of magnitude speed degradation when considering the results in Table VII; although processing would still be fast, with the worst case time for a 192 byte payload being just over 21ms for the slowest mode. However, to know whether this processing is fast enough, or the bottleneck for the protocol, we need to also consider the communication speed via the smart card to Point of Sale (POS) interface.

VII. Communication Effects on Performance

Performance tests of AE, normally just focus on the processing aspects, as communication in an Internet-connected world is generally fast enough (e.g., 25-100Mbps) to cause negligible delay. However, for payment card use of AE we are dealing with interfaces that may be *much* slower and so transactions might hit communication limits before card processing limits.

A. Payment Card Interfaces

The interfaces for payment cards fall into two main categories. The contact interface is the oldest and has dominated payment card transactions using Chip & PIN, however many cards now support the contactless interface for touch and pay (no PIN). Within the standards (contact [8] and contactless [9]) a range of interface speeds are defined, however this does not mean the fastest modes are supported in all deployed cards, or POS terminals. Table VIII shows an example range of transmission speeds and an estimation of the time to transmit the data associated with the different sized test messages. Note that the working interface speed is negotiated and agreed between the smart card and the POS terminal as part of the pre-transaction protocol and by varying clock speed as well as divider parameters the full range would be closer to 9600 - 38400 bits/s. For example the contact rates in Table VIII are computed in accordance with standards, as a clock frequency (5 MHz) f_c divided by factor D (372, 512 and 512 respectively) and multiplied by a factor F (1, 8 and 32 respectively).

The speed range is very wide especially in the contact case, as the default rates maintain compatibility with very old cards and POS terminals. The command processing and transmission can be considered as separate activities; and whichever takes longer is considered the bottleneck limit. Recalling the MULTOS platform performance (Table III) we have a processing limited solution. There are some message/mode combinations that are communications limited, but only when running at the lowest default speed, which is impractically slow. If we now recall the raw native mode results (Table VII), then in practice we have a communications limited solution. At the fastest interface speeds this may not be quite the case, however we would not normally assume that the fastest rates would be available from cards and POS terminals; and so the 78,125 bps and 106,000 bps for contact and contactless interfaces respectively would be more reasonable expectations. The future outlook is that the communication rates will get faster and the contact interface will eventually be displaced by contactless, which suggests that transactions will be processing limited. EMV implementations in mobile phones will of course have access to much faster wireless technologies such as 802.11ac that can run at 1.3 Gbits/s, however the scope of this study is restricted to conventional smart card devices.

VIII. Conclusions

The study investigated AE modes on existing available smart chips/platforms using conventional crypto-coprocessors. GCM was not analysed in detail as the *multH* function (or parts of it) would need to be implemented within more specialist crypto-coprocessor hardware. All the other AE modes considered, were feasible both in terms of speed and memory usage. The native mode implementation was much faster than the MULTOS platform and in the final tests all the modes for all single APDU test message sizes took no more than 2.14ms.

The new results differ markedly from previous comparisons that have focussed on general processors, larger message sizes and the inclusion of Associated Data. The native ETM/CCM modes were quicker than OCB for the single APDU test messages although OCB modes would be expected to claw back the advantage for multi-APDU messages. In our native implementation, and for a single APDU, ETM was always slightly ahead of CCM and OCB2 led OCB3.

At first glance the results may seem counter-intuitive due to the extra encryptions required in ETM/CCM compared to OCB2/OCB3, however they arise because the chip has significant crypto-coprocessor gain. The native measurements show that the core DES encryption time is comparable with a 16 byte block XOR executed by the CPU. We suggested a new benchmark, the Technology Independent Gain Assessment (TIGA) for CPUs with crypto-coprocessors; as the percentage of the block encryption that can be completed by the crypto-coprocessor in the time it would take the CPU to compute a block XOR. We estimated that the MULTOS platform and native chip had TIGAs of 22% and 100% (33% for TDES) respectively. The new TIGA measure could be valuable when comparing algorithm implementations on various platform types, as may increasingly be the case in Internet of Things implementations.

The performance gain from the crypto-coprocessor can be eroded if more time is spent conditioning the data into and out of it. Such processing may be required for security protection, (to mask data and/or to reduce leakage), although it should be noted that any part of an algorithm running in the CPU may also require similar protection.

The processing time comparison was independent of the communications interface speed, however both affect the overall protocol performance. The MULTOS platform is primarily

 ISBN: 978-1-61208-547-0

processing limited, whereas the simple native implementation is mainly communications limited. If we degrade the native performance by an order of magnitude in anticipation of overheads to reduce side-channel leakage (e.g., repeated operations or hamming weight equalisation in software) then we approach the optimum around the 78,125bps rate; any lower than this and the protocol performance will degrade due to communication delays.

The crypto-coprocessor gain, coupled with small message sizes, means that there is not much to choose between OCB2, OCB3, ETM and CCM performance. It might be argued that ETM could be chosen for speed and efficiency of small-/medium messages or OCB if medium/large messages are the norm. It is also possible for GCM to be usable in future if supported by a specialist co-processor, however it is unlikely to be much quicker than the other modes. As performance is unlikely to be a great differentiator for the AE modes, an option could be to standardise an AE framework around a default mode and define a negotiation process for a card and POS terminal to agree alternative AE modes. This would provide a useful mechanism if vulnerabilities were discovered in any particular AE mode, as well as a means for interworking and migration of smart cards and POS terminals having different capabilities.

A. Future Work

It would be interesting to implement the AE modes in a similar manner on other secured microcontrollers with crypto-coprocessors (although this may be difficult due to publication restrictions required by device vendors). In the first instance this should help prove the generality of the results, but also provide more evidence on the usefulness of the TIGA benchmark, which is easily determined on any processor. It is hoped that a secured smart card microcontroller chip could become available (for academic research) offering native mode programming and crypto-coprocessor support for GCM, so that a full-set of AE mode results could be generated and published. A Java Card platform has become available that would permit direct comparison with the MULTOS platform, as both are based on the SLE78 secured microcontroller.

REFERENCES

[1] D. Boneh, R. Demillo, and R. Lipton, “On the importance of checking computations”, in *Advances in Cryptography - Eurocrypt 97*, volume 1233, pp. 37-51, Springer Verlag, 2013.

[2] CC, ”Common criteria for information technology security evaluation part1: Introduction and general model,” version 3.1 release 4, September 2012.

[3] EMV, “Books 1-4,” Version 4.3, 2011.

[4] EMVCo, http://www.emvco.com/ [retrieved: March, 2017].

[5] FIPS, “Federal information Processing Standards, Data Encryption Standard (DES), publication 46-3” http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf [retrieved: March, 2017].

[6] FIPS, “Federal Information Processing Standards, Announcing the Advanced Encryption Standard (AES), Publication 197.” http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf [retrieved: March, 2017].

[7] Infineon, ”SLE78CAFX4000P(M) short product overview,” v11.12, 2012.

[8] ISO/IEC, “7816 identification cards - integrated circuit(s) cards with contacts,” parts 1-4, 1999.

[9] ISO/IEC, “14443 identification cards - contactless integrated circuit cards - proximity cards,” parts 1-4, 2008.

[10] ISO/IEC, “19772 Information technology - Security techniques - Authenticated encryption,” 2009.

[11] ISO/IEC, “9797 Information technology - Security techniques - Message Authentication Codes (MACs),” parts 1-3, 2011.

[12] P. Kocher, “Timing attacks on implementations of diffie-hellman RSA DSS and other systems,” in *Advances in Cryptology - CRYPTO ’96 Proceedings LNCS*, volume 1109, pp. 104-113 Springer Verlag, 1996.

[13] P. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” in *Advances in Cryptology - Crypto 99 Proceedings LNCS*, volume 1666, pp. 388-397, Springer Verlag, 1999.

[14] T. Krovetz and P. Rogaway, “The software performance of authenticated encryption modes, fast software encryption, RFC 7253,” in *FSE 2011 Proceedings*, pp. 306-327, Springer verlag, 2011.

[15] T. Krovetz and P. Rogaway, “The OCB authenticated-encryption algorithm, IETF RFC 7253,” May 2014.

[16] D. McGrew and J. Viega, “The galois/counter mode of operation (GCM),” parts 1-3, May 2005, http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf [retrieved: March, 2017].

[17] MULTOS, http://www.multos.com/ [retrieved: March, 2017].

[18] MULTOS, “The MULTOS developer’s reference manual,” MAO-DOC-TEC-006 v1.49, 2013.

[19] NIST, “Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality, SP800-38C,” May 2004.

[20] NIST, “Recommendation for block cipher modes of operation: Galois/-counter mode (GCM) and GMAC, SP800-38D,” November 2007.

[21] P. Rogaway, “OCB mode,” http://web.cs.ucdavis.edu/˜rogaway/ocb/ [retrieved: March, 2017].

[22] J. Salowey, A. Choudhury, and D. McGrew, ”AES galois counter mode (GCM) cipher suites for TLS, IETF RFC 5288,” August 2008.

[23] Samsung, “S3CC9E4/8: 16-bit CMOS microcontroller for smart card user’s manual,” rev 0, 2004.

[24] D. Whiting, R. Housley, and N. Ferguson, ”Counter with CBC-MAC (CCM), IETF RFC 3610,” September 2003.

 ISBN: 978-1-61208-547-0