Comb to Pipeline: Fast Software Encryption Revisited[†]

Andrey Bogdanov, Martin M. Lauridsen, and Elmar Tischhauser

DTU Compute, Technical University of Denmark, Denmark
{anbog,mmeh,ewti}@dtu.dk

Abstract. AES-NI, or Advanced Encryption Standard New Instructions, is an extension of the x86 architecture proposed by Intel in 2008. With a pipelined implementation utilizing AES-NI, parallelizable modes such as AES-CTR become extremely efficient. However, out of the four non-trivial NIST-recommended encryption modes, three are inherently sequential: CBC, CFB, and OFB. This inhibits the advantage of using AES-NI significantly. Similar observations apply to CMAC, CCM and a great deal of other modes. We address this issue by proposing the *comb scheduler* – a fast scheduling algorithm based on an efficient look-ahead strategy, featuring a low overhead – with which sequential modes profit from the AES-NI pipeline in real-world settings by filling it with multiple, independent messages.

As our main target platform we apply the comb scheduler to implementations on Haswell, a recent Intel microarchitecture, for a wide range of modes. We observe a *drastic speed-up of factor* 5 *for NIST's CBC, CFB, OFB and CMAC* performing around 0.88 cpb. Surprisingly, contrary to the entire body of previous performance analysis, the *throughput of the authenticated encryption (AE) mode CCM gets very close to that of GCM and OCB3*, with about 1.64 cpb (vs. 1.63 cpb and 1.51 cpb, resp.), when message lengths are sampled according to a realistic distribution for Internet packets, despite Haswell's heavily improved binary field multiplication. This suggests CCM as an AE mode of choice as it is NIST-recommended, does not have any weak-key issues like GCM, and is royalty-free as opposed to OCB3. Among the CAESAR contestants, the comb scheduler significantly speeds up CLOC/SILC, JAMBU, and POET, with the mostly sequential nonce-misuse resistant design of POET, performing at 2.14 cpb, becoming faster than the well-parallelizable COPA. Despite Haswell being the target platform, we also include performance figures for the more recent Skylake microarchitecture, which provides further optimizations to AES-NI instructions.

Finally, this paper provides the first optimized AES-NI implementations for the novel AE modes OTR, CLOC/SILC, COBRA, POET, McOE-G, and Julius.

Keywords. AES-NI, `pclmulqdq`, Haswell, Skylake, authenticated encryption, CAESAR, CBC, OFB, CFB, CMAC, CCM, GCM, OCB3, OTR, CLOC, COBRA, JAMBU, SILC, McOE-G, COPA, POET, Julius

[†] This is an extended version of [9] which appeared at FSE 2015

1 Introduction

With the introduction of AES-NI, Advanced Encryption Standard New Instructions, on Intel's microarchitectures starting from Westmere and later as well as on a variety of AMD CPUs, AES received a sigfinicant speed-up in standard software, going well below 1 cycle per byte (cpb) and possessing a constant running time, which also thwarts cache-timing attacks. Important applications for AES-NI include OpenSSL, Microsoft's BitLocker, Apple's FileVault, TrueCrypt, PGP and many more. In a nutshell, AES-NI provides dedicated instructions for AES encryption and decryption. In this work we focus on Intel's very recent Haswell architecture as our target platform. However, since the proceedings version of this paper [9], Intel has released its latest architecture codenamed Skylake, which further improves the AES-NI instructions. For this reason, we give also an overview of the performance of the modes considered on this platform (see Table 1b below).

On Haswell, the latency of the AES-NI instructions is 7 clock cycles (cc) and the throughput is 1 cc. That is, AES-NI has a pipeline of length 7 and one can issue one instruction per clock cycle. This pipeline can be naturally exploited by parallel AES modes such as CTR in the encryption domain, PMAC in the message authentication domain as well as GCM and OCB in the authenticated encryption domain.

However, numerous AES modes of operation – both standardized and novel such as CAESAR[1] submissions – are essentially sequential by design. Indeed, NIST-standardized CBC, CFB, OFB and CMAC [12] as well as CLOC and POET from FSE 2014 and McOE-G from FSE 2012 are essentially sequential, which limits their performance on state-of-the-art servers and desktops significantly, as the pipeline cannot be filled entirely, having a severe performance penalty as a consequence.

In this paper, we aim to address this gap and propose an *efficient look-ahead comb scheduler for real-world Internet packets*. Its application can change the landscape of AES modes of operation in terms of their practical throughput. Our contributions are as follows:

Novel Comb Scheduler. Communication devices of high-speed links are likely to process many messages at the same time. Indeed, on the Internet, the bulk of data is transmitted in packets of sizes between 1 and 2 KB, following a bimodal distribution. While most previous implementations of block cipher modes consider processing a single message, we propose to process several messages in parallel, which reflects this reality. This is particularly beneficial when using an inherently sequential mode. In this work, for the first time, we deal with AES modes of operation in this setting (see Section 3). More specifically, as our main contribution, we propose an efficient look-ahead comb scheduler. For real-world packet lengths on the Internet, this algorithm allows us to fill the pipeline of

[1] Competition for Authenticated Encryption: Security, Applicability, and Robustness.

AES-NI and attain significant speed-ups for many popular modes. After covering some background in Section 2, we present our comb scheduler and its analysis in Section 3.

Speed-up of factor 5 for NIST's CBC, OFB, CFB and CMAC. When applied to the NIST-recommended encryption and MAC modes, our comb scheduler delivers a performance gain on Haswell of a factor 5 with the real-world packet sizes. The modes get as fast as 0.88 cpb compared to around 4.5 cpb in the sequential message processing setting. These results are provided in Section 4.

Change of landscape for AE When our comb scheduler is applied to AE modes of operation, a high performance improvement is attained as well with the real-world message size distribution. CCM, having a sequential CBC-based MAC inside, gets as fast as GCM and OCB which are inherently parallel. Being royalty-free, NIST-recommended and weak-key free, CCM becomes an attractive AE mode of operation in this setting.

In the context of the ongoing CAESAR competition, in the domain of nonce-misuse resistant modes, the essentially sequential POET gets a significant speed-up of factor 2.7 down to 2.14 cpb when implemented on Haswell. Its rival CAESAR contestant COPA runs as 2.68 cpb, while being insecure under release of unverified plaintext. This is somewhat surprising, considering that POET uses 3 AES calls per block vs. 2 AES calls per block for COPA.

Section 5 also contains first-time comprehensive performance evaluations for the Haswell platform of further AES-based modes in the CAESAR competition and beyond, both in the sequential and comb-scheduled implementations, including OTR, CLOC/SILC, JAMBU, COBRA, McOE-G and Julius.

Faster $GF(2^{128})$ multiplications on Haswell Section 6 focuses on the technical implementation tricks on Haswell that we used to obtain our results and contains a detailed study of improved $GF(2^{128})$ multiplications on the architecture.

Sample performance Table 1a gives a performance summary for the Haswell microarchitecture (our target platform in this work), of the NIST-recommended modes of operation which, among others, we benchmark in this work. The figures in the table represent different scenarios for message lengths in both the case of sequential processing on one hand, and when our proposed comb scheduler is used on the other hand. In Table 1b we present an overview of the performance of all modes considered in this work on the more recent Intel microarchitecture codenamed Skylake. Again, the numbers are presented for both sequential processing, and when using our comb scheduler, where applicable.

Table 1: Sample performance for various block cipher modes of operation considered in this work, using sequential processing on one hand, and the proposed comb scheduler algorithm for message processing on the other hand. Fixed lengths refer to messages of 2 KB in size, while realistic lengths refer to a sample according to actual packet lengths on the Internet (see Section 3). All numbers are in cycles/byte.

(a) Performance of NIST-recommended modes of operation on Intel's Haswell microarchitecture (i5-4300U CPU running at 1.90GHz).

	Fixed lengths		Realistic lengths	
Mode	Sequential	**Comb**	Sequential	**Comb**
AES-ECB	0.63	—	0.65	—
AES-CTR	0.74	—	0.78	—
AES-CBC	4.38	**0.65**	4.47	**0.87**
AES-OFB	4.39	**0.67**	4.48	**0.88**
AES-CFB	4.36	**0.65**	4.45	**0.89**
CMAC-AES	4.35	**0.64**	4.29	**0.84**
CCM	5.10	**1.37**	5.22	**1.64**

(b) Performance of all modes considered in this work on the more recent Skylake microarchitecture (i7-6700 CPU running at 3.40GHz).

	Fixed lengths		Realistic lengths	
Mode	Sequential	**Comb**	Sequential	**Comb**
ECB	0.63	—	0.64	—
CBC	2.65	**0.64**	2.08	
CTR	0.63	—	0.67	—
OFB	2.68	**0.65**	2.11	
CFB	2.63	**0.64**	2.07	
CMAC	2.67	**0.64**	2.15	**0.78**
Julius	2.58	—	2.91	—
McOE-G	5.23	**1.39**	5.28	**1.53**
JAMBU	5.52	**1.53**	5.61	**1.69**
SILC	2.82	**1.28**	2.80	**1.48**
POE	2.95	**1.92**	3.60	**2.10**
OTR	0.87	—	1.36	—
COPA	1.69	—	2.18	—
GCM	0.71	—	1.16	—
CCM	3.36	**1.27**	3.41	**1.43**
CLOC	2.82	**1.26**	2.80	**1.44**
OCB3	0.70	—	1.27	—
COBRA	2.67	—	2.87	—

2 Background

In this paper, we consider AES-based symmetric primitives, that is, algorithms that make use of the (full) AES block cipher in a black-box fashion. In particular, this includes block cipher modes of operation, block cipher based message authentication codes, and authentication encryption (AE) modes.

NIST-recommended Modes. In its special publications SP-800-38A-D [12], NIST recommends the following modes of operation: ECB, CBC, CFB, OFB and CTR as basic encryption modes; CMAC as authentication mode; and CCM and GCM as authenticated encryption modes.

Authenticated Encryption Modes and CAESAR. Besides the widely employed and standardized modes CCM and GCM, a great number of modes for authenticated encryption have been proposed, many of them being contestants in the currently ongoing CAESAR competition. We give a brief overview of the AE modes we will consider in this study.

We split up our consideration into two categories: (i) *nonce-misuse resistant* AE modes, by which we mean modes that maintain authenticity and privacy up to a common message prefix even when the nonce is repeated (also called OAE security) and (ii) *nonce-based* AE modes which either lose authenticity, privacy or both when nonces are repeated. The modes we consider in the former camp are McOE-G, COPA, POET and Julius, while the nonce-based modes considered are CCM, GCM, OCB3, OTR, CLOC, COBRA, JAMBU and SILC. Table 2 gives a comparison of the modes considered in this work. The price to pay for a mode to be nonce-misuse resistant includes extra computation, a higher serialization degree, or both. One of the fundamental questions we answer in this work is how much one has to pay, in terms of performance, to maintain this level of security when repeating nonces.

For the specifications of the AE modes considered, we refer to the relevant references listed in Table 2. We clarify that for COBRA we refer to the FSE 2014 version with its reduced security claims (compared to the withdrawn CAESAR candidate); with POET we refer to the version where the universal hashing is implemented as full AES-128 (since using four rounds would not comprise a mode of operation); and with Julius, we mean the CAESAR candidate regular Julius-ECB.

The AES-NI Instruction Set. Proposed in 2008 and implemented as of their 2010 Westmere microarchitecture, Intel developed special instructions for fast AES encryption and decryption [17], called the *AES New Instruction Set (AES-NI)*. It provides instructions for computing one AES round `aesenc, aesenclast`, its inverse `aesdec, aesdeclast`, and auxiliary instructions for key scheduling. The instructions do not only offer better performance, but security as well, since they are leaking no timing information. AES-NI is supported in a subset of

Table 2: Overview of the AE modes considered in this paper. The ‖ column indicates parallelizability; the "IF" column indicates whether a mode needs the inverse of the underlying block cipher in decryption/verification; the "E" and "M" columns give the number of calls, per message block, to the underlying block cipher and multiplications in $GF(2^n)$, respectively.

	Ref.	Year	‖	IF	E	M	Description
Nonce-based AE modes							
CCM	[39]	2002	–	yes	2	–	CTR encryption, CBC-MAC authentication
GCM	[33]	2004	yes	yes	1	1	CTR mode with chain of multiplications
OCB3	[28]	2010	yes	–	1	–	Gray code-based xor-encrypt-xor (XEX)
OTR	[35]	2013	yes	yes	1	–	Two-block Feistel structure
CLOC	[23]	2014	–	yes	1	–	CFB mode with low overhead
COBRA	[5]	2014	yes	yes	1	1	Combining OTR with chain of multiplications
JAMBU	[40]	2014	–	yes	1	–	AES in stream mode, lightweight
SILC	[24]	2014	–	yes	1	–	CLOC with smaller hardware footprint
Nonce-misuse resistant AE modes							
McOE-G	[13]	2011	–	–	1	1	Serial multiplication-encryption chain
COPA	[4]	2013	yes	–	2	–	Two-round XEX
POET	[1]	2014	yes	–	3	–	XEX with two AXU (full AES-128 call) chains
Julius	[7]	2014	–	–	1	2	SIV with polynomial hashing

Westmere, Sandy Bridge, Ivy Bridge and Haswell microarchitectures. A range of AMD processors also support the instructions under the name *AES Instructions*, including processors in the Bulldozer, Piledriver and Jaguar series [21].

Pipelining. Instruction pipelines allow CPUs to execute the same instruction for data-independent instances in an overlapping fashion. This is done by subdividing the instruction into steps called *pipeline stages*, with each stage processing its part of one instruction at a time. The performance of a pipelined instruction is characterized by its latency L (number of cycles to complete one instruction) and throughput T (the number of cycles to wait between issuing instructions). For instance, on the original Westmere architecture, the AES-NI `aesenc` instruction has a latency of 6 cycles and a throughput of 2, meaning that one instruction can be issued every two cycles.

Previous Work. Matsui and Fukuda at FSE 2005 [31] and Matsui [30] at FSE 2006 pioneered comprehensive studys on how to optimize symmetric primitives on the then-contemporary generation of Intel microprocessors. One year later, Matsui and Nakajima [32] demonstrated that the vector instruction units of the Core 2 architecture lends itself to very fast bitsliced implementations of block ciphers. For the AES, on a variety of platforms, Bernstein and Schwabe [8] developed various micro-optimizations yielding vastly improved performance.

Intel's AES instructions were introduced to the symmetric community by Shay Gueron's tutorial [16] at FSE 2009. In the same year, Käsper and Schwabe announced new records for bitsliced AES-CTR and AES-GCM performance [27]. At FSE 2010, Osvik et al. [37] explored fast AES implementations on AVR and GPU platforms. Finally, a study of the performance of CCM, GCM, OCB3 and CTR modes was presented by Krovetz and Rogaway [28] at FSE 2011.

3 Comb Scheduler: An Efficient Look-Ahead Strategy

3.1 Motivation

A substantial number of block cipher modes of operation for (authenticated) encryption are inherently sequential in nature. Among the NIST-recommended modes, this includes the classic CBC, OFB, CFB and CCM modes as well as CBC derivatives such as CMAC. Also, more recent designs essentially owe their sequential nature to design goals, e.g allowing lightweight implementations or achieving stricter notions of security, for instance not requiring a nonce for security (or allowing its reuse). Examples of such include ALE [10], APE [3], CLOC [23] the McOE family of algorithms [13,14], and some variants of POET [1]. While being able to perform well in other environments, such algorithms cannot benefit from the available pipelining opportunities on contemporary general-purpose CPUs. For instance, as detailed in Section 6, the AES-NI encryption instructions on Intel's recent Haswell architecture feature a high throughput of $T = 1$ instruction/cycle, but a relatively high latency of $L = 7$ cycles. Modes of operation that need to process data sequentially will invariably be penalized in such environments.

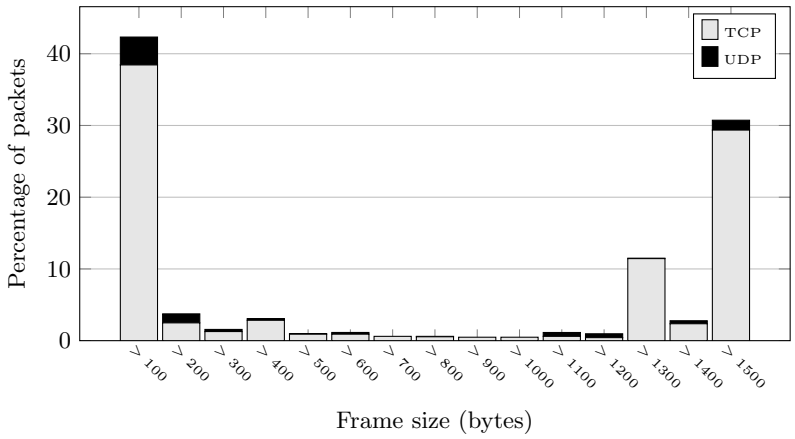


Fig. 1: Distribution of frame sizes for TCP and UDP

Furthermore, even if designed with parallelizability in mind, (authenticated) modes of operation for block ciphers typically achieve their best performance when operating on somewhat longer messages, often due to the simple fact that these diminish the impact of potentially costly initialization phases and tag generation. Equally importantly, only longer messages allow high-performance software implementations to make full use of the available pipelining opportunities [2, 18, 28, 34]. In practice, however, one rarely encounters messages which allow to achieve the maximum performance of an algorithm. Recent studies on packet sizes on the Internet demonstrate that they basically follow a bimodal distribution [26, 36, 38]: 44% of packets are between 40 and 100 bytes long; 37% are between 1400 and 1500 bytes in size; the remaining 19% are somewhere in between. Throughout the paper, we refer to this as the *realistic* distribution of message lengths. A distribution of frame sizes in TCP and UDP from [36] is shown in Figure 1. This emphasizes the importance of good performance for messages up to around 2 KB, as opposed to longer messages. Second, when looking at the weighted distribution, this implies that the vast majority of data is actually transmitted in packets of medium size between 1 and 2 KB. Considering the first mode of the distribution, we remark that many of the very small packets of Internet traffic comprise TCP ACKs (which are typically not encrypted), and that the use of authentication and encryption layers such as TLS or IPsec incurs overhead significant enough to blow up a payload of 1 byte to a 124 byte packet [22]. It is therefore this range of message sizes (128 to 2048 bytes) that authenticated modes of encryption should excel at processing, when employed for encryption of Internet traffic.

3.2 Filling the Pipeline: Multiple Messages

It follows from the discussion above that the standard approach of considering one message at a time, while arguably optimizing message processing latency, can not always generate optimal throughput in high-performance software implementations in most practically relevant scenarios. This is not surprising for the inherently sequential modes, but even when employing a parallelizable design, the prevailing distribution of message lengths makes it hard to achieve the best performance. In order to remedy this, we propose to consider the scheduling of multiple messages in parallel *already in the implementation of the algorithm itself*, as opposed to considering it as a (single-message) black box to the message scheduler. This opens up possibilities of increasing the performance in the cases of both sequential modes and the availability of multiple shorter or medium-sized messages. In the first case, the performance penalty of sequential execution can potentially be hidden by filling the pipeline with a sufficient number of operations on independent data. In the second case, there is a potential to increase performance by keeping the pipeline filled also for the overhead operations such as block cipher or multiplication calls during initialization or tag generation.

Note that while in this paper we consider the processing of multiple messages on a single core, the multiple message approach naturally extends to multi-core settings. Conceptually, the transition of a sequential to a multiple message

implementation can be viewed as similar to the transition from a straightforward to a bit-sliced implementation approach. We note also, that an idealistic view of multiple-message processing was given in [10] for the dedicated authenticated encryption algorithm ALE. This consideration was rather rudimentary, did not involve real-world packet size distributions, and did not treat any modes of operation. It is also important to note that while multiple message processing has the potential to increase the throughput of an implementation, it can also increase its latency (see also Section 3.4). The degree of parallelism therefore has to be chosen carefully and with the required application profile in mind.

3.3 Message Scheduling with a Comb

Consider the scenario where a number of messages of varying lengths need to be processed by a sequential encryption algorithm. As outlined before, blocks from multiple messages have to be processed in an interleaved fashion in order to make use of the available inter-message parallelism. Having messages of different lengths implies that generally the pipeline cannot always be filled completely. At the same time, the goal to schedule the message blocks such that pipeline usage is maximized has to be weighed against the computational cost of making such scheduling decisions: in particular, every conditional statement during the processing of the bulk data results in a pipeline stall.

Algorithm 1: CombScheduler

Input : k messages $M_1, \ldots, M_k$ of lengths $\ell_1, \ldots, \ell_k$ blocks
Input : Parallelism degree P

$L \leftarrow$ list of tuples (M_j, ℓ_j) sorted by decreasing ℓ_j
Denote by $L[i] = (M_i, \ell_i)$ the i-th tuple in L
while $|L| > 0$ **do**
  $r \leftarrow \min\{P, |L|\}$
  Perform initialization for messages $M_1, \ldots, M_r$
  $\mathcal{P}, \mathcal{B} \leftarrow$ PrecomputeWindows$(\ell_1, \ldots, \ell_r)$
  $completedBlocks \leftarrow 0$
  for $w = 1, \ldots, |\mathcal{P}|$ **do** *// Loop over windows*
    for $i = 1, \ldots, \mathcal{B}[w]$ **do** *// Loop over blocks in window*
      for $j = 1, \ldots, \mathcal{P}[w]$ **do** *// Loop over messages in window*
        Process block $(completedBlocks + i)$ of message M_j
      end
    end
    $completedBlocks \leftarrow completedBlocks + \mathcal{B}[w]$
  end
  Perform finalization for messages $M_1, \ldots, M_r$
  Remove the r first elements from L
end

In order to reconcile the goal of exploiting multi-message parallelism for sequential algorithms with the need for low-overhead scheduling, we propose *comb*

scheduling. Comb scheduling is based on the observation that ideally, messages processed in parallel have the same length, so given a desired (maximum) parallelism degree P and a list of message lengths $\ell_1, \ldots, \ell_k$, we can subdivide the computation in a number of *windows*, in each of which we process as many consecutive message blocks as we can in so-called *windows*, for as many independent messages as possible, according to the restrictions based on the given message lengths. Since our scheduling problem exhibits optimal substructure, this greedy approach yields an optimal solution. Furthermore, the scheduling decisions of how many blocks are to be processed at which parallelism level can be precomputed once the ℓ_i are known. This implies that instead of making each processing step conditional, we only have conditional statements whenever we proceed from one group to the next. The comb scheduling method is outlined in Algorithms 1 and 2.

Algorithm 2: PRECOMPUTEWINDOWS($\ell_1, \ldots, \ell_r$)

Input : r message lengths $\ell_1, \ldots, \ell_r$ in blocks, s.t. $\forall i = 1, \ldots, r-1 : \ell_i \geq \ell_{i+1}$
Output: List $\mathcal{P}$ with $\mathcal{P}[w]$ the number of messages to process in parallel in window w
Output: List $\mathcal{B}$ with $\mathcal{B}[w]$ the number of blocks to process in window w

$\mathcal{P} \leftarrow []$, $\mathcal{B} \leftarrow []$ *// Initialize to empty lists*
$w \leftarrow 1$, $q_{last} \leftarrow 0$, $i \leftarrow r$
while $i > 1$ **do** *// Scan windows right to left*
  $q \leftarrow \ell_i$, $j \leftarrow i - 1$
  while $j \geq 1$ *and* $\ell_j = \ell_i$ **do** $j \leftarrow j - 1$; *// Left-extend while lengths equal*
  $\mathcal{P}[w] \leftarrow i$
  $\mathcal{B}[w] \leftarrow q - q_{last}$
  $q_{last} \leftarrow q$, $i \leftarrow j$, $w \leftarrow w + 1$
end
if $i = 1$ **then** *// Leftover message*
  $\mathcal{P}[w] \leftarrow 1$
  $\mathcal{B}[w] \leftarrow \ell_1 - q_{last}$
end
return $\mathcal{P}, \mathcal{B}$

In order to simplify the combing, the messages are pre-sorted by decreasing length. This sorting step can be implemented via an optimal sorting network for the constant value of P chosen by the implementation, and can employ pointer swapping only, without copying of data blocks. Alternatively, a low-overhead algorithm like *insertion sort* can be used. The sorted messages are then processed in groups of P. A pre-computation is performed to determine the windows inside the group, i.e. how many windows are required to process the group, and for each window, how many messages still have blocks left to be processed (and how many blocks need processing in the windows). This information is returned in the lists $\mathcal{P}$and $\mathcal{B}$by Algorithm 2. Inside each group, the processing is window by window according to the precomputed parallelism levels $\mathcal{P}$ and window lengths

$\mathcal{B}$: in window w, the same $\mathcal{P}[w]$ messages of the current message group are processed $\mathcal{B}[w]$ blocks further. In the next window, at least one message will be exhausted, and the parallelism level decreases by at least one. As comb scheduling is processing the blocks by common (sub-)length from left to right, our method can be considered a symmetric-key variant of the well-known comb method for (multi-)exponentiation [29].

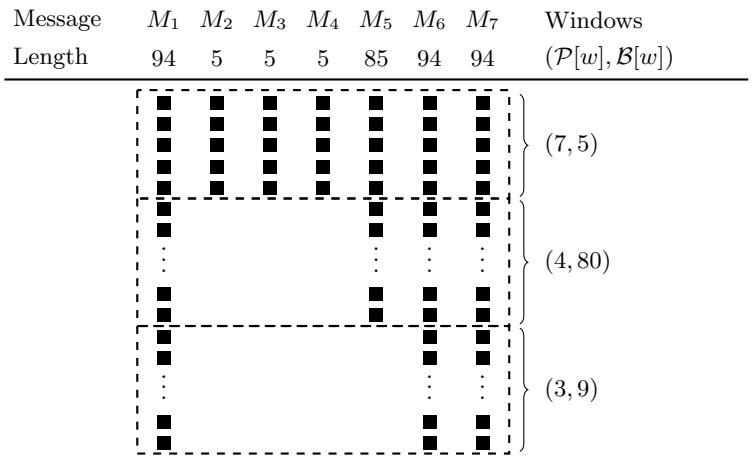


Fig. 2: Comb scheduling example for 7 messages of lengths $(\ell_1, \ldots, \ell_7) = (94, 5, 5, 5, 85, 94, 94)$ blocks

An Example. We illustrate comb scheduling in Figure 2 with an example where $P = k = 7$: The pre-computation determines that all $\mathcal{P}[1] = 7$ messages can be processed in a pipelined fashion for the first $\mathcal{B}[1] = 5$ blocks; $\mathcal{P}[2] = 4$ of the 7 messages can be processed further for the next $\mathcal{B}[2] = 80$ blocks; and finally $\mathcal{P}[3] = 3$ remaining messages are processed for another $\mathcal{B}[3] = 9$ blocks.

Choice of Parallelism Degree. In order to make optimal use of the pipeline, the parallelism degree P should be chosen according to

$$P = L \cdot T,$$

with L denoting the latency (in cycles) and T the throughput (in instructions/cycle) of the pipelined instruction. For AES-NI, the latency and throughput of the `aesenc` instruction vary from platform to platform. A summary for the Haswell microarchitecture is given in Table 8 in Section 6.2, suggesting $P = 7$ for this platform.

3.4 Latency vs. Throughput

A point worth discussing is the latency increase one has to pay when using multiple message processing. Since the speed-up is limited by the parallelization level, one can at most hope for the same latency as in the sequential processing case. We illustrate this by the example of CBC mode when implemented in the multiple message setting with comb scheduling. We consider two distributions for message lenghts: one where all messages are 2048 bytes long, and one realistic distribution of Internet traffic. The performance data is given in Table 3.

Table 3: Performance of CBC encryption (in cpb) and relative speed-up for comb scheduling with different parallelization levels for fixed message lengths of 2048 bytes (top) and realistic message lengths (bottom).

		Parallelization level P						
	Sequential	2	3	4	5	6	7	8
2K messages	4.38	2.19	1.47	1.11	0.91	0.76	0.66	0.65
Relative speed-up	×1.00	×2.00	×2.98	×3.95	×4.81	×5.76	×6.64	×6.74
Realistic distribution	4.38	2.42	1.73	1.37	1.08	0.98	0.87	0.85
Relative speed-up	×1.00	×1.81	×2.53	×3.20	×4.06	×4.47	×5.03	×5.15

What we can see from Table 3 is, that for messages of an identical length of 2 KB, the ideal linear speed-up of a factor P is actually achieved for $P \in \{2, 3, 4\}$ parallel messages: setting $|M| = 2048$, instead of waiting $4.38 \cdot |M|$ cycles in the sequential case, one has a latency of either $2.19 \cdot 2 = 4.38 \cdot |M|$ cycles when $P = 2$; when $P = 3$ the latency is $1.47 \cdot 3 = 4.41 \cdot |M|$ cycles; and when $P = 4$ the latency is $1.11 \cdot 4 = 4.44 \cdot |M|$ cycles. Starting from $P = 5$ parallel messages, the latency slightly increases with the throughput, however remaining at a manageable level even for $P = 7$ parallel messages, where it is only around 5% higher than in the sequential case, while achieving a 6.64 times increase in throughput. For realistic message lengths, using $P = 7$ multiple messages, we see an average increase in latency of 39% which has to be contrasted to (and, depending on the application, weighed against) the significant 5.03 times increase in throughput.

4 Pipelined NIST-recommended Modes

In this section, we present the results of our performance study of the NIST-recommended encryption- and MAC modes when instantiated with AES as the block cipher, and implemented with AES-NI and AVX vector instructions. Reminding that some modes covered, such as CBC and CFB, are sequential in encryption but parallel in decryption, we remark that we only benchmark encryption in this work.

Experimental Setting. All measurements were taken on a single core of an Intel Core i5-4300U CPU (Haswell) at 1900 MHz. For each combination of parameters, the performance was determined as the median of 91 averaged timings of 200 measurements each. This method has also been used by Krovetz and Rogaway in their benchmarking of authenticated encryption modes in [28]. The measurements are taken over samples from the realistic distribution on message lengths.

Out of the basic NIST-recommended modes, ECB and CTR are inherently parallelizable and already achieve good performance with trivial sequential message scheduling. Three other modes, CBC, OFB and CFB, however, are inherently sequential and therefore need to make use of inter-message parallelism to benefit from the available pipelining. The same holds for the NIST-recommended CMAC message authentication code. We therefore measure the performance of all modes with sequential processing, and additionally the performance of the sequential modes with comb message scheduling.

Table 4: Performance comparison (in cpb) of NIST-recommended encryption- and MAC modes, with trivial sequential processing and comb scheduling. Message lengths are sampled from the realistic Internet traffic distribution.

Mode	Sequential processing	**Comb scheduling**	Speed-up
AES-ECB	0.65	—	—
AES-CTR	0.78	—	—
AES-CBC	4.47	**0.87**	×5.14
AES-OFB	4.48	**0.88**	×5.09
AES-CFB	4.45	**0.89**	×5.00
CMAC-AES	4.29	**0.84**	×5.10

Discussion. Our performance results for pipelined implementations of NIST-recommended modes are presented in Table 4. It is apparent that the parallel processing of multiple messages using comb scheduling speeds up encryption performance by a factor of around 5, bringing the sequential modes within about 10% of CTR mode performance. The results also indicate that the overhead induced by the comb scheduling algorithm itself can be considered negligible compared to the AES calls.

Due to their simple structure with almost no overhead, it comes as no surprise that CBC, OFB and CFB performance are virtually identical. That CMAC performs slightly better despite additional initialization overhead can be explained by the fact that there are no ciphertext blocks to be stored to memory.

5 Pipelined Authenticated Encryption

We now turn our attention to the AES-NI software performance of authenticated encryption modes. We consider the well-established modes CCM, GCM and OCB3 as well as a number of more recent proposals, many of them being contestants in the ongoing CAESAR competition.

Experimental Setting. The same experimental setup as for the NIST-recommended modes above applies. For our performance measurements, we are interested in the performance of the various AE modes of operation during their *bulk processing* of message blocks, i.e. during the encryption phase. To that end, we *do not* measure cycles spent on processing associated data. As some schemes can have a significant overhead when computing authentication tags (finalization) for short messages, we *do* include this phase in the measurements as well.

5.1 Performance in the Real World

Out of the AE modes in consideration, GCM, OCB3, OTR, COBRA, COPA and Julius are parallelizable designs. We therefore only measure their performance with sequential message processing. On the other hand, CCM, CLOC, SILC, JAMBU, McOE-G and POET are sequential designs and as such will also be measured in combination with comb scheduling. In all cases, we again measure the performance using message lengths sampled from the realistic bimodal distribution of typical Internet traffic.

Table 5 lists the results of the performance measurements. For the parallelizable modes where comb scheduling was implemented, the relative speed-up compared to normal sequential processing is indicated in the last column. In this table, the nonce-based AE modes are listed separately from those offering nonce-misuse resistance in order to provide a better estimation of the performance penalty one has to pay for achieving a stricter notion of security.

Discussion. The performance data demonstrates that comb scheduling of multiple messages consistently provides a speed-up of factors between 3 and 4 compared to normal sequential processing. For typical Internet packet sizes, comb scheduling enables sequential AE modes to run with performance comparable to the parallelizable designs, in some cases even outperforming them. This can be attributed to the fact that AE modes typically have heavier initialization and finalization than normal encryption modes, consisting of setting up variables and generating the authentication tag, both implying a penalty in performance for short messages. By using comb scheduling, however, also the initial and final AES calls can be (at least partially) parallelized between different messages. The relative speed-up for this will typically reduce with the message length. The surprisingly good performance of McOE-G is due to the fact that it basically benefits doubly from multiple message processing: not only the AES calls, but also its sequential finite field multiplications can now be pipelined. For the comb

Table 5: Performance comparison (in cpb) of AES-based AE modes with trivial sequential processing and comb scheduling. Message lengths are sampled from the realistic Internet traffic distribution. Proposals from the CAESAR competition are marked by a †.

(a) Nonce-based AE modes

Mode	Sequential	**Comb**	Speed-up
CCM	5.22	**1.64**	×3.18
GCM	1.63	—	—
OCB3†	1.51	—	—
OTR†	1.91	—	—
COBRA	3.56	—	—
CLOC†	4.47	**1.45**	×3.08
JAMBU†	9.12	**2.05**	×4.45
SILC†	4.53	**1.49**	×3.04

(b) Nonce-misuse resistant AE modes

Mode	Sequential	**Comb**	Speed-up
McOE-G	7.41	**1.79**	×4.14
COPA†	2.68	—	—
POET†	5.85	**2.14**	×2.73
Julius†	3.73	—	—

scheduling implementation of CCM, which is two-pass, it is worth noting that all scheduling precomputations only need to be done once, since exactly the same processing windows can be used for both passes.

Best Performance Characteristics. From Table 5, it is apparent that for encryption of typical Internet packets, the difference between sequential and parallelizable modes, with respect to performance, somewhat blurs when comb scheduling is employed. This is especially true for the nonce-based setting, where CLOC, SILC, CCM, GCM and OCB3 all perform on a very comparable level. For the nonce-misuse resistant modes, our results surprisingly show better performance of the two sequential modes for this application scenario. This can be attributed to the fact that the additional processing needed for achieving nonce-misuse resistance hampers performance on short messages, which can be mitigated to some extent by comb scheduling.

5.2 Traditional Approach: Sequential Messages of Fixed Lengths

While the previous section analyzed the performance of the various AE modes using a model for a realistic message lengths, we provide some more detail on the exact performance exhibited by these modes for a range of (fixed) message lengths in this section. To that end, we provide performance measurements for specific message lengths between 128 and 2048 bytes. The results are summarized in Table 6.

Discussion. The performance data clearly shows the expected difference between sequential and parallelizable modes when no use of multiple parallel messages can be made. Among the sequential modes, only initialization-heavy modes

Table 6: Performance comparison (in cpb) of AE modes for processing a single message of various, fixed message lengths.

(a) Nonce-based modes

	Message length (bytes)				
Mode	128	256	512	1024	2048
CCM	5.35	5.19	5.14	5.11	5.10
GCM	2.09	1.61	1.34	1.20	1.14
OCB3	2.19	1.43	1.06	0.87	0.81
OTR	2.97	1.34	1.13	1.02	0.96
CLOC	4.50	4.46	4.44	4.46	4.44
COBRA	4.41	3.21	2.96	2.83	2.77
JAMBU	9.33	9.09	8.97	8.94	8.88
SILC	4.57	4.54	4.52	4.51	4.50

(b) Nonce-misuse resistant modes

	Message length (bytes)				
Mode	128	256	512	1024	2048
McOE-G	7.77	7.36	7.17	7.07	7.02
COPA	3.37	2.64	2.27	2.08	1.88
POET	6.89	5.74	5.17	4.88	4.74
Julius	4.18	4.69	3.24	3.08	3.03

such as McOE-G and POET show significant performance differences between shorter and longer messages, while this effect usually is very pronounced for the parallelizable modes. It can be seen from Table 6, that for the nonce-based modes, the best performance is generally offered by OCB3, although OTR and GCM provide quite similar performance on Haswell. Among the nonce-misuse resistant modes, COPA has the best performance for all message sizes.

5.3 Exploring the Limits: Upper Bounding the Combing Advantage

Having seen the performance data with comb scheduling for realistic message lengths, it is natural to consider the question what the performance of the various modes would be for the ideal scenario where the scheduler is given only messages of a fixed length. In this case, the comb precomputation would result in only one processing window, so essentially no scheduler-induced branches are needed during the processing of the messages. In a sense, this constitutes an *upper bound* for the multi-message performance with comb scheduling for the various encryption algorithms.

Table 7 summarizes the performance of the previously considered sequential AE modes when comb scheduling is combined with fixed message lengths.

Discussion. It can be seen that for all modes considered, the performance for longer messages at least slightly improves compared to the realistic message length mix of Table 5, although the differences are quite small and do not exceed around 0.2 cpb. For shorter messages, the difference can be more pronounced for a mode with heavy initialization such as POET. Overall, this shows that comb scheduling for a realistic distribution provides a performance which is very comparable to that of comb scheduling of messages with an idealized distribution.

Table 7: Performance comparison (in cpb) of sequential AE modes when comb scheduling is used for various fixed message lengths.

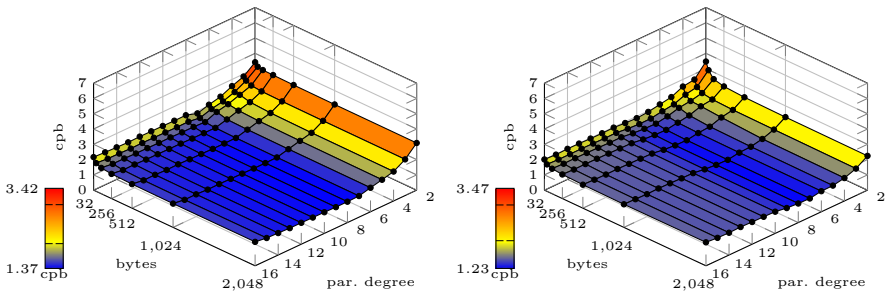
(a) Nonce-based modes

	Message length (bytes)				
Mode	128	256	512	1024	2048
CCM	1.51	1.44	1.40	1.38	1.37
CLOC	1.40	1.31	1.26	1.24	1.23
JAMBU	2.14	1.98	1.89	1.85	1.82
SILC	1.43	1.33	1.28	1.25	1.24

(b) Nonce-misuse resistant modes

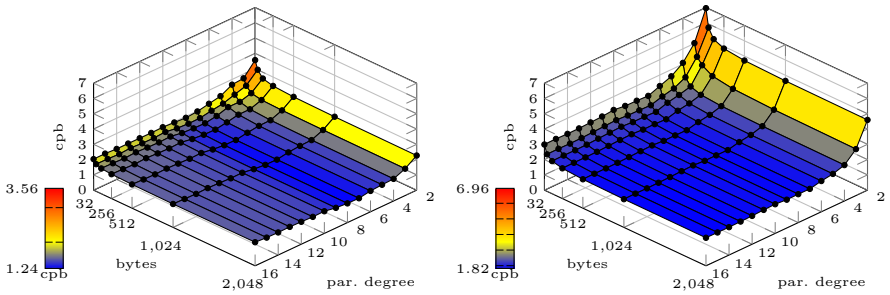
	Message length (bytes)				
Mode	128	256	512	1024	2048
McOE-G	1.91	1.76	1.68	1.64	1.62
POET	2.56	2.23	2.06	1.97	1.93

Exploring the Parameter Space. Besides the distribution of the message lengths, the parallelization degree influences the performance of the comb scheduler. Even though $P = 7$ is optimal for Haswell, applications might choose a lower value if typically only few messages are available simultaneously, in order to avoid a latency blowup. The dependency of the performance on both individual parameters is further detailed in Figures 3 and 4, where the comb scheduling performance is shown for a range of fixed message lengths ranging from 32 bytes to 2048 bytes, and parallelization degrees $P \in \{2, \ldots, 16\}$. The horizontal lines in the color key of each plot indicate the integer values in the interval.



(a) CCM

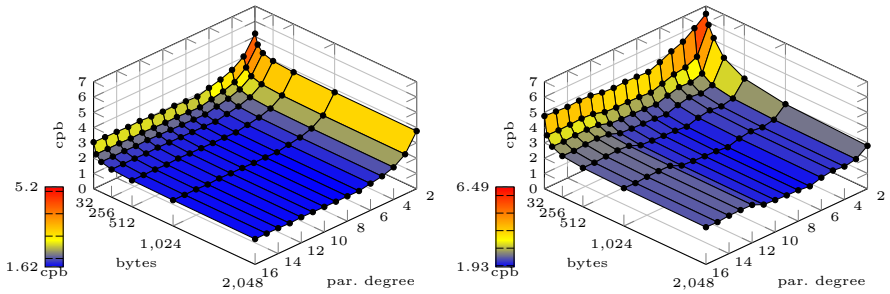
(b) CLOC



(c) SILC

(d) JAMBU

Fig. 3: Performance of serial nonce-based AE modes of operation when comb scheduling is used with different parallelization levels for various fixed message lengths



(a) McOE-G

(b) POET

Fig. 4: Performance of serial nonce-misuse resistant AE modes of operation when comb scheduling is used with different parallelization levels for various fixed message lengths

Impact of Working Set Sizes. It can be seen from the plots that, as expected, most modes achieve their best speed-up in the multiple messages scenario for a parallelization level of around 7 messages. It is worth noting, however, that for each of these messages, a complete working set, i.e. the internal state of the algorithm, has to be maintained. Since only sixteen 128-bit `xmm` registers are available in Haswell, even a working set of three 128-bit words (for instance cipher state, tweak mask, checksum) for 7 simultaneously processed messages will already exceed the number of available registers. As the parallelization degree P increases, the influence of this factor increases. This can be seen especially for POET, which has a larger internal state per instance. By contrast, CCM, JAMBU and McOE-G suffer a lot less from this effect.

The experimental results also confirm the intuition of Section 6.1 that Haswell's improved memory interface can handle fairly large working set sizes efficiently by hiding the stack access latency between the cryptographic operations. This allows more multiple messages to be processed faster despite the increased register pressure, basically until the number of moves exceeds the latency of the other operations, or ultimately the limits of the Level-1 cache are reached.

6 Haswell Tricks: Towards Faster Code

In this section, we describe some of the optimization techniques and architecture features that were used for our implementations on Haswell.

6.1 General Considerations: AVX and AVX2 Instructions

In our Haswell-optimized AE scheme implementations we make heavy use of Intel Advanced Vector Extensions (AVX) which has been present in Intel processors since the Sandy Bridge microarchitecture. AVX can be considered as an extension of the SSE+[2] streaming SIMD instructions operating on 128-bit `xmm0` through `xmm15` registers. While AVX and AVX2, the latter which appears first on Haswell, brings mainly support for 256-bit wide registers to the table, this is not immediately useful in implementing a AES-based modes, as the AES-NI instructions as well as the `pclmulqdq` instruction support only the use of 128-bit `xmm` registers. However, a feature of AVX that we use extensively is the three-operand enhancement, due to the VEX coding scheme, of legacy two-operand SSE2 instructions. This means that, in a single instruction, one can non-destructively perform binary vector operations on two operands and store the result in a third operand, rather than overwriting one of the inputs, e.g. one can do $c = a \oplus b$ rather than $a = a \oplus b$. This eliminates overhead associated with `mov` operations required when overwriting an operand is not acceptable. With AVX, three-operand versions of the AES-NI and `pclmulqdq` instructions are also available.

A further Haswell feature worth taking into account is the increased throughput for logical instructions such as `vpxor/vpand/vpor` on AVX registers: While

[2] i.e. SSE, SSE2, etc.

the latency remains at one cycle, now up to 3 such instructions can be scheduled simultaneously. Notable exceptions are algorithms heavily relying on mixed 64/128 bit logical operations such as JAMBU, for which the inclusion of a fourth 64-bit ALU implies that such algorithms will actually benefit from frequent conversion to 64-bit arithmetic via the `vpextrq/vpinsrq` instructions, rather than artificial extension of 64-bit operands to 128 bits for operation on the AVX registers.

On Haswell, the improved memory controller allows two simultaneous 16-byte aligned moves `vmovdqa` from registers to memory, with a latency of one cycle. This implies that on Haswell, the comparatively large latency of cryptographic instructions such as `vaesenc` or `pclmulqdq` allows the implementer to "hide" more memory accesses to the stack when larger internal state of the algorithm leads to register shortage. This also aids the generally larger working sets induced by the multiple message strategy described in Section 3.

6.2 Improved AES Instructions

In Haswell, the AES-NI encryption and decryption instructions had their latency improved from $L = 8$ cycles on Sandy and Ivy Bridge[3], down to $L = 7$ cycles [20]. This is especially beneficial for sequential modes such as AES-CBC, CCM, McOE-G, CLOC, SILC and JAMBU. Furthermore, the throughput has been slightly optimized, allowing for better performance in parallel. Table 8 gives an overview of the latencies and inverse throughputs measured on our test machine (Core i5-4300U). The data was obtained using the test suite of Fog [15].

Table 8: Experimental latency L (cycles) and inverse throughput T^{-1} (cycles/instruction) of AES-NI and `pclmulqdq` instructions on Intel's Haswell microarchitecture

Instruction	L	T^{-1}
`aesenc`	7	1
`aesdec`	7	1
`aesenclast`	7	1
`aesdeclast`	7	1
`aesimc`	14	2
`aeskeygenassist`	10	8
`pclmulqdq`	7	2

6.3 Improvements for Multiplication in $GF(2^{128})$

The `pclmulqdq` instruction was introduced by Intel along with the AES-NI instructions [19], but is not part of AES-NI itself. The instruction takes two 128-bit

[3] We remark that Fog reports a latency of 4 cycles for `aesenc` on Ivy Bridge [15]

inputs and a byte input `imm8`, and performs carry-less multiplication of a combination of one 64-bit half of each operand. The choice of halves of the two operands to be multiplied is determined by the value of bits 4 and 0 of `imm8`.

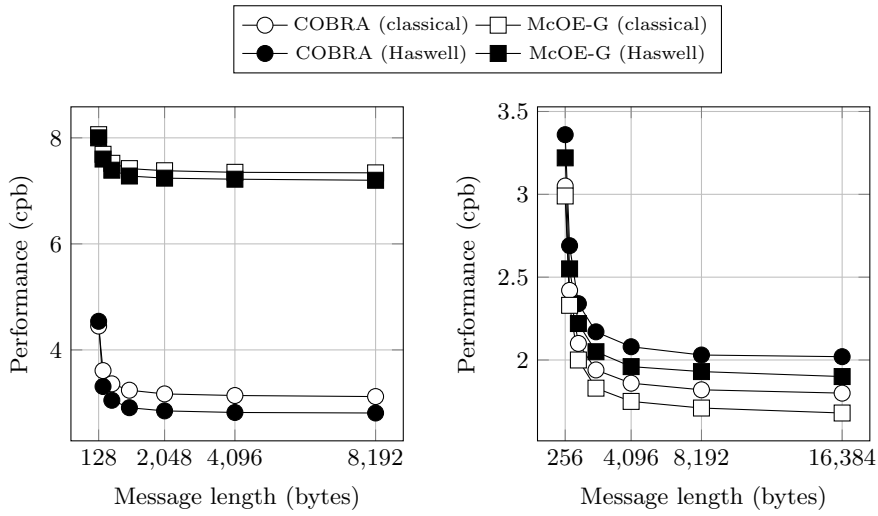
Most practically used AE modes employing multiplication in a finite field use block lengths of 128 bits. As a consequence, multiplications are in the field $GF(2^{128})$. As the particular choice of finite field does not influence the security proofs, modes use the tried-and-true GCM finite field. For our performance study, we have used two different implementation approaches for finite field multiplication, which we in general denote `gfmul`. The first implementation, which we refer to as the *classical method*, was introduced in Intel's white paper [19]. It applies `pclmulqdq` three times in a carry-less Karatsuba multiplication followed by modular reduction. The second implementation variant, which we refer to as the *Haswell-optimized method*, was proposed by Gueron [18] with the goal of leveraging the much improved `pclmulqdq` performance on Haswell to trade many shifts and XORs for one more multiplication. This is motivated by the improvements in both latency (7 vs. 14 cycles) and inverse throughput (2 vs. 8 cycles) on Haswell [20].

In modes where the output of a multiplication over $GF(2^{128})$ is not directly used, other than as a part of a chain combined using addition, the aggregated reduction method by Jankowski and Laurent [25] can be used to gain speed-ups. This method uses the inductive definitions of chaining values combined with the distributivity law for the finite field to postpone modular reduction at the cost of storing powers of an operand. Among the modes we benchmark in this work, the aggregated reduction method is applicable only to GCM and Julius. We therefore use this approach for those two modes, but apply the general `gfmul` implementations to the other modes.

6.4 Classical vs. Haswell $GF(2^{128})$ Multiplication

Here we compare the classical and Haswell-optimized methods of multiplication in $GF(2^{128})$. We compare the performance of the AE modes considered that use full $GF(2^{128})$ multiplications (as opposed to aggregated reduction): McOE-G and COBRA, when instantiated using the two different multiplication methods. Figure 5 shows that when processing a single message, the Haswell-optimized method performs better than the classical implementation of `gfmul`, while the situation is the other way around, when processing multiple messages in parallel.

Considering the optimizations made for the `pclmulqdq` instruction on Haswell, these observations make perfect sense. When processing only a single message, there is no independent data available on which to draw parallelism. As such, and since the finite field multiplication in COBRA and McOE-G is sequential, this becomes a bottleneck for single message processing, and the optimizations made to the instruction come to their right. On the other hand, when processing multiple messages, there is enough independent data to draw on to keep the pipeline filled, so the latency improvement of the instruction vanishes, and in turn the four instruction calls for the Haswell multiplication method aggravate the overall latency.



(a) Processing a single message

(b) Processing multiple messages

Fig. 5: Performance of COBRA and McOE-G using the classical- and Haswell multiplication methods for a single message (left) and 8 multiple messages of equal length (right)

6.5 Haswell-optimized Doubling in $GF(2^{128})$

The doubling operation in $GF(2^{128})$ is commonly used in AE schemes [6], and indeed among the schemes we benchmark, it is used by OCB3, OTR, COBRA, COPA and POET. Doubling in this field consists of left shifting the input by one bit and doing a conditional XOR of a reduction polynomial if the most significant bit of the input equals one. Neither SSE+ nor AVX provide an instruction to shift a full `xmm` register bitwise, nor to directly test only its most significant bit. As such, these functions have to be emulated with other operations, opening up a number of implementation choices.

Listing 1.1: Doubling in $GF(2^{128})$

```
1  __m128i xtime(__m128i v) {
2    __m128i v1, v2;
3    v1 = _mm_slli_epi64(v, 1);
4    v2 = _mm_slli_si128(v, 8);
5    v2 = _mm_srli_epi64(v2, 63);
6    if (msb of v == 1)
7      return _mm_xor_si128
8             (_mm_or_si128(v1, v2
                ), RP);
9    else
10     return _mm_or_si128(v1, v2)
            ;
11 }
```

Table 9: Performance of doubling with different approaches to MSB testing

	Approach	Cycles
(i)	Extraction	15.4
(ii)	Test	15.4
(iii)	MSB mask	16.7
(iv)	Compare + extract	5.6

We emulate a left shift by one bit by the following procedure, which is optimal with regard to the number of instructions and cycles: given an input v, the value $2v \in GF(2^{128})$ is computed as in Listing 1.1. Consider $v = (v_L \| v_R)$ where v_L and v_R are 64-bit values. In line 3 we set $v_1 = (v_L \ll 1 \| v_R \ll 1)$ and lines 4 and 5 set first $v_2 = (v_R \| 0)$ and then $v_2 = ((v_R \gg 63) \| 0)$. As such, we have $v \ll 1 = v_1 \mid v_2$. This leaves us with a number of possibilities when implementing the branching of line 6, which can be categorized as (i) extracting parts from v and testing, (ii) AVX variants of the `test` instruction, (iii) extracting a mask with the most significant bit of each part of v and (iv) comparing against a mask $\texttt{MSB\_MASK} = \texttt{80}\cdots\texttt{00}$ and then extracting from the comparison result. Some of these approaches again leave several possibilities regarding the number of bits extracted, etc.

Interestingly, the approach taken to check the most significant bit of v has a substantial impact on the doubling performance. This is illustrated by Table 9 where we give performance of the doubling operation using various combinations of approaches. The numbers are obtained by averaging over 10^8 experiments. Surprisingly, we see that there is a significant speed-up, about a factor $\times 3$, when using comparison with `MSB_MASK` combined with extraction, over the other methods. Thus, we suggest to use this approach, where line 6 can be implemented as

if (_mm_extract_epi8(_mm_cmpgt_epi8(MSB_MASK, v), 15) == 0).

7 Conclusions

In this paper, we have discussed the performance of various block cipher-based symmetric primitives when instantiated with the AES on Intel's recent Haswell architecture.

As a general technique to speed up both inherently sequential modes and to deal with the typical scenario of having many shorter messages, we proposed our *comb scheduler*, an efficient algorithm for the scheduling of multiple simultaneous messages which is based on a look-ahead strategy within a certain window size. This leads to significant speed-ups for essentially all sequential modes, even when taking realistic Internet traffic distributions into account. Applied to the NIST-recommended modes CBC, CFB, OFB and CMAC, comb scheduling attains a significant speed-up of factor at least 5, resulting in a performance of around 0.88 cpb, which is within about 10% of the performance of the parallelizable CTR mode on the same message distribution.

Applying comb scheduling to authenticated encryption modes (which typically feature higher initialization and finalization overhead, thus penalizing performance on the frequently occurring short messages), our technique speeds up the inherently sequential AE modes CCM, CLOC, SILC, JAMBU, McOE-G and POET by factors between 3 and 4.5. This particularly results in a CCM performance comparable to GCM or OCB3, without being afflicted by issues with weak-key classes or encumbered by patents.

Our study also establishes that for practitioners wishing to use a nonce-misuse resistant AE mode, the POET design with comb scheduling attains better performance than the completely parallelizable mode COPA. Since POET furthermore offers ciphertext-misuse resistance, this suggests that users do not have to choose between good performance or stricter notions of security.

References

1. Farzaneh Abed, Scott R. Fluhrer, Christian Forler, Eik List, Stefan Lucks, David A. McGrew, and Jakob Wenzel. Pipelineable on-line encryption. In Cid and Rechberger [11], pages 205–223.
2. Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. *Breakthrough AES Performance with Intel AES New Instructions*. Intel Corporation, 2010.
3. Elena Andreeva, Begl Bilgin, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha, and Kan Yasuda. APE: authenticated permutation-based encryption for lightweight cryptography. In Cid and Rechberger [11], pages 168–186.
4. Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Elmar Tischhauser, and Kan Yasuda. Parallelizable and authenticated online ciphers. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part I*, volume 8269 of *Lecture Notes in Computer Science*, pages 424–443. Springer, 2013.
5. Elena Andreeva, Atul Luykx, Bart Mennink, and Kan Yasuda. COBRA: A parallelizable authenticated online cipher without block cipher inverse. In Cid and Rechberger [11], pages 187–204.
6. Kazumaro Aoki, Tetsu Iwata, and Kan Yasuda. How Fast Can a Two-Pass Mode Go? A Parallel Deterministic Authenticated Encryption Mode for AES-NI. In *DIAC 2012: Directions in Authenticated Ciphers*, 2012.
7. Lear Bahack. Julius: Secure Mode of Operation for Authenticated Encryption Based on ECB and Finite Field Multiplications.
8. Daniel J. Bernstein and Peter Schwabe. New AES software speed records. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *Progress in Cryptology - INDOCRYPT 2008, 9th International Conference on Cryptology in India, Kharagpur, India, December 14-17, 2008. Proceedings*, volume 5365 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2008.
9. Andrey Bogdanov, Martin M. Lauridsen, and Elmar Tischhauser. Comb to pipeline: Fast software encryption revisited. In Gregor Leander, editor, *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, volume 9054 of *Lecture Notes in Computer Science*, pages 150–171. Springer, 2015.
10. Andrey Bogdanov, Florian Mendel, Francesco Regazzoni, Vincent Rijmen, and Elmar Tischhauser. ALE: AES-based Lightweight Authenticated Encryption. In Shiho Moriai, editor, *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*, volume 8424 of *Lecture Notes in Computer Science*, pages 447–466. Springer, 2013.
11. Carlos Cid and Christian Rechberger, editors. *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, volume 8540 of *Lecture Notes in Computer Science*. Springer, 2015.

12. Morris J. Dworkin. SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Technical report, Gaithersburg, MD, United States, 2007.
13. Ewan Fleischmann, Christian Forler, and Stefan Lucks. Mcoe: A family of almost foolproof on-line authenticated encryption schemes. In Anne Canteaut, editor, *Fast Software Encryption - 19th International Workshop, FSE 2012, Washington, DC, USA, March 19-21, 2012. Revised Selected Papers*, volume 7549 of *Lecture Notes in Computer Science*, pages 196–215. Springer, 2012.
14. Ewan Fleischmann, Christian Forler, Stefan Lucks, and Jakob Wenzel. McOE: A Family of Almost Foolproof On-Line Authenticated Encryption Schemes. Cryptology ePrint Archive, Report 2011/644, 2011. `http://eprint.iacr.org/`.
15. Agner Fog. Software Optimization Resources. Accessed on February 17, 2014. `http://www.agner.org/optimize/`, February 2014.
16. Shay Gueron. Intel's new AES instructions for enhanced performance and security. In Orr Dunkelman, editor, *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, volume 5665 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2009.
17. Shay Gueron. *Intel Advanced Encryption Standard (AES) New Instructions Set*. Intel Corporation, 2010.
18. Shay Gueron. AES-GCM software performance on the current high end CPUs as a performance baseline for CAESAR. In *DIAC 2013: Directions in Authenticated Ciphers*, 2013.
19. Shay Gueron and Michael E. Kounavis. *Intel Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode*. Intel Corporation, 2010.
20. Sean Gulley and Vinodh Gopal. *Haswell Cryptographic Performance*. Intel Corporation, 2013.
21. Brent Hollingsworth. *New "Bulldozer" and "Piledriver" Instructions*. Advanced Micro Devices, Inc., 2012.
22. Steven Iveson. IPSec Bandwidth Overhead Using AES. Accessed on February 17, 2014. `http://packetpushers.net/ipsec-bandwidth-overhead-using-aes/`, October 2013.
23. Tetsu Iwata, Kazuhiko Minematsu, Jian Guo, and Sumio Morioka. CLOC: authenticated encryption for short input. In Cid and Rechberger [11], pages 149–167.
24. Tetsu Iwata, Kazuhiko Minematsu, Jian Guo, Sumio Morioka, and Eita Kobayashi. SILC: SImple Lightweight CFB.
25. Krzysztof Jankowski and Pierre Laurent. Packed AES-GCM Algorithm Suitable for AES/PCLMULQDQ Instructions. pages 135–138, 2011.
26. Wolfgang John and Sven Tafvelin. Analysis of internet backbone traffic and header anomalies observed. In *Internet Measurement Conference*, pages 111–116, 2007.
27. Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009.
28. Ted Krovetz and Phillip Rogaway. The Software Performance of Authenticated-Encryption Modes. In Antoine Joux, editor, *Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers*, volume 6733 of *Lecture Notes in Computer Science*, pages 306–327. Springer, 2011.

29. Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In Yvo Desmedt, editor, *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 1994.
30. Mitsuru Matsui. How far can we go on the x64 processors? In Matthew J. B. Robshaw, editor, *Fast Software Encryption, 13th International Workshop, FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers*, volume 4047 of *Lecture Notes in Computer Science*, pages 341–358. Springer, 2006.
31. Mitsuru Matsui and Sayaka Fukuda. How to maximize software performance of symmetric primitives on pentium III and 4 processors. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.
32. Mitsuru Matsui and Junko Nakajima. On the power of bitslice implementation on intel core2 processor. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 121–134. Springer, 2007.
33. David A. McGrew and John Viega. The Galois/Counter Mode of Operation (GCM).
34. David A. McGrew and John Viega. The Security and Performance of the Galois/-Counter Mode (GCM) of Operation. In Anne Canteaut and Kapalee Viswanathan, editors, *Progress in Cryptology - INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20-22, 2004, Proceedings*, volume 3348 of *Lecture Notes in Computer Science*, pages 343–355. Springer, 2004.
35. Kazuhiko Minematsu. Parallelizable Rate-1 Authenticated Encryption from Pseudorandom Functions. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 275–292. Springer, 2014.
36. David Murray and Terry Koziniec. The state of enterprise network traffic in 2012. In *Communications (APCC), 2012 18th Asia-Pacific Conference on*, pages 179–184. IEEE, 2012.
37. Dag Arne Osvik, Joppe W. Bos, Deian Stefan, and David Canright. Fast software AES encryption. In Seokhie Hong and Tetsu Iwata, editors, *Fast Software Encryption, 17th International Workshop, FSE 2010, Seoul, Korea, February 7-10, 2010, Revised Selected Papers*, volume 6147 of *Lecture Notes in Computer Science*, pages 75–93. Springer, 2010.
38. Kostas Pentikousis and Hussein G. Badr. Quantifying the deployment of TCP options - a comparative study. pages 647–649, 2004.
39. Doug Whiting, Russ Housley, and Niels Ferguson. Counter with CBC-MAC (CCM), 2003.
40. Hongjun Wu and Tao Huang. JAMBU Lightweight Authenticated Encryption Mode and AES-JAMBU.