White Paper

Jim Guilford
Sean Gulley
Erdinc Ozturk
Kirk Yap
Vinodh Gopal
Wajdi Feghali

IA Architects
Intel Corporation

Fast Multi-buffer IPsec Implementations on Intel® Architecture Processors

December 2012

328332-001

Executive Summary

This paper describes the Intel® Multi-Buffer Crypto for IPsec Library, a family of highly-optimized software implementations of the core cryptographic processing for IPsec, which provides industry-leading performance on a range of Intel® Processors.

This paper describes the usage of the IPsec library and presents a summary of the performance for some algorithm pairs. We can achieve a single-thread throughput performance of ~14 Gigabits/second on an Intel® Core™ i7 processor 2600, for AES-128 encryption in the CBC-XCBC mode.[1]

[1] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Configurations: Refer to the Performance section on page 16. For more information go to http://www.intel.com/performance.

Contents

Overview 4
Background of IPsec 4
Supported Algorithms 5
APIs 5
Multi-buffer API 5
Basic API 5
Integration into an Application 7
Flushing 7
Job structure 9
Pre-expanded AES Keys 11
HMAC IPad and OPad 12
AES XCBC Precomputes 13
Selecting a Set of Functions 14
GCM API 15
Building 15
Performance 16
Methodology 16
Results 17
Conclusion 19
Contributors 20
References 20

Overview

This paper describes the Intel® Multi-Buffer Crypto for IPsec Library [5], a set of functions that implement the computationally intensive authentication and encryption algorithms for IPsec. These functions provide an easy way for an IPsec implementation to take advantage of the benefits of multi-buffer processing.

This paper assumes that the reader is at least somewhat familiar with Intel's Multi-buffer processing. If not, the reader may want to read [1] first for background.

Background of IPsec

Internet Protocol Security (IPsec) is a suite of protocols for securing internet traffic using the Internet Protocol (IP). Two of the most computationally intensive operations on the bulk data within IPsec are encryption and authentication.

IPsec is embedded in the IP stack in a number of implementations, for example within Linux. Once a connection is established and data is flowing, a significant number of CPU cycles is spent in encrypting or decrypting the bulk data, and in computing a cryptographic hash (MAC) of the data in order to validate its authenticity.

We've previously shown [1] that multi-buffer processing can significantly speed up processing in many cases. The IPsec functions described in this paper extend that work to handling combined encryption and authentication using a variety different underlying algorithms.

Supported Algorithms

This version of the library supports the following cryptographic and hash algorithms (for both encryption and decryption):

Encryption	Authentication
AES-128 CBC AES-192 CBC AES-256 CBC AES-128 CTR AES-192 CTR AES-256 CTR	HMAC SHA-1 HMAC SHA-224 HMAC SHA-256 HMAC SHA-384 HMAC SHA-512 HMAC MD5 AES-128-XCBC
AES-128 GCM[2]	

APIs

There are two independent sets of APIs in the associated code [5]. One handles multi-buffer processing for packets requiring AES and HMAC processing. It is primarily with this interface that this paper is concerned. There is an independent set of APIs for GCM processing. This code is the same as described in [2] and separately released.

Multi-buffer API

The multi-buffer API is essentially an extension of the API described in [1]. One "theme" of the interface is to pre-compute data that is likely to be shared between many packets, so that it does not need to be recalculated multiple times. These calculations will be described in detail later.

Basic API

The basic API exists in three forms, with one version using the SSE instruction set, one using AVX, and one using AVX2. Each of the following functions exists in three forms, one with the suffix "_sse", one with "_avx", and one with "_avx2". In the following discussion, the functions will use the suffix "_xxx" to represent one of the above.

Note that the data structures are independent of the suffix; however they are initialized differently based on the suffix. Thus, one cannot mix different suffixes when using the same multi-buffer manager object.

---

[2] Not Multi-buffered

The functions are summarized below:

init_mb_mgr_xxx	Initialize the MB_MGR state object
get_next_job_xxx	Get a new job object
submit_job_xxx	Submit the job that was previously gotten
flush_job_xxx	Return the oldest job object
get_completed_job_xxx	Return the oldest job object only if is already completed

The basic idea is that the application needs to provide multiple jobs before the previous jobs complete their processing. This can be called an "asynchronous" interface. The application does this by submitting jobs to the multi-buffer manager (MB_MGR). For every job that it submits, it may receive a completed job, or it may receive NULL. In general, if a job is returned, it will not be the one that was just submitted. However, jobs will be returned **in the same order** that they were submitted.

These routines are **not** thread-safe. If they are being called by multiple threads, then the application must take care that calls are not made from different threads at the same time, i.e. thread-safety should be implemented at a level higher than these routines. These routines do not make operating-system calls, and in particular they do not allocate memory.

In general, there will be an arbitrary number of jobs that have been submitted, but which have not yet been returned, and are therefore "outstanding". To avoid having the application manage this arbitrary number of job objects, the management of the job objects is handled by the MB_MGR. The application gets a pointer to the next available job object by calling get_next_job_xxx(). The application then fills in the job data fields appropriately, and then submits it by calling submit_job_xxx(). If this returns a non-NULL job, then that job has been completed (unless its arguments are invalid) and the application should do whatever it needs to in order to finish processing that job.

The returned job object is not explicitly returned to the MB_MGR. Rather, it is implicitly returned by the next call to get_next_job_xxx(). Another way to put this is that the returned job object may be referenced until the next call to get_next_job_xxx(). After this, it is no longer safe to access the previous job's fields.

One measure of job latency is the number of submit_job_xxx() calls that must be made before the submitted job is returned. Since jobs are returned in order, and at most one job is returned for every job submitted, this "latency" can never decrease; it can only stay the same or increase. To allow the latency to decrease, there is an optional function that may be called, get_completed_job_xxx(). This will return the next job *if it was already completed*. If the next job is not yet completed, no processing will be done, and this function will return NULL.

The usage of these functions may be illustrated by the following pseudo-code:

```
init_mb_mgr_xxx(&mb_mgr);
...
while (work_to_be_done) {
    job = get_next_job_xxx(&mb_mgr);
    // TODO: Fill in job fields
    job = submit_job_xxx(&mb_mgr);
    while (job) {
        // TODO: Complete processing on job
        job = get_completed_job(&mb_mgr);
    }
}
```

Integration into an Application

In general, how this library is integrated into an application depends on the design of the application and is beyond the scope of this paper, but here are some approaches.

One main issue is how to accumulate multiple jobs without blocking, waiting for the jobs to finish. In the best case, there is already an asynchronous interface, either providing a stream of jobs, or perhaps providing a work-queue containing jobs, which can feed the library.

In other designs, there may be many threads, where each thread wants to submit a job and then block until that job completes. One way to deal with this is to have each thread enqueue its job into a thread-safe queue, and then to have a compute thread pull jobs off of this queue and process them.

Alternately, each thread could take a mutex, submit its job, signal the returned job (if any) as complete, and then release the mutex and wait for its job to be so signaled.

Note that the library is designed to fully utilize the core, so there is no performance to be gained by having two instances of the library running on the same processor.

There are probably many other designs or architectures that one could use to interface the sources of jobs with the multi-buffer manager.

Flushing

Using the API described in the previous section, when the stream of incoming jobs ends, there is no way to get back the remaining outstanding jobs. That functionality is provided by flush_job_xxx(). This is similar to submit_job_xxx() except that no new job is submitted, and that a completed job will always be returned unless there are no outstanding jobs.

Note that a "flushed" job is completed normally; i.e. it is correctly and fully processed. The flush_job_xxx() function is different from get_completed_job_xxx() in that flushing will, in general, perform algorithmic processing, and will always return the oldest job unless there are no outstanding jobs; whereas get_completed_job_xxx() will never perform algorithmic processing, and will only return the oldest job if it was completed in a previous function call.

Flushing is more expensive than submitting in that the system is less efficient when flushing than when submitting. So, for example, one could use the library by always calling "flush" after every "submit". This would result in correct behavior, but the performance would be worse than if one used well-implemented single-buffer code. The presumption of the multi-buffer code is that flushing will occur much less often than submitting.

A typical reason to use flushing is to deal with a lull in incoming jobs. Imagine that there was a steady stream of incoming jobs, but then for a short period of time there were no new jobs. In the absence of flushing, the last jobs submitted before the lull would not be returned until after the lull, when more new jobs appeared. This would result in an unreasonably long latency for these jobs. In this case, flushing can be used to complete these remaining jobs before new jobs arrive.

In a sense, the concept of submitting vs. flushing is that when jobs are coming at a rapid rate, they are all submitted, and the multi-buffer efficiency is high. When jobs are arriving at a slow rate or not at all, then flushing is invoked, which reduces efficiency. But since the jobs are coming at a slow rate, the overall system can tolerate a lower efficiency.

Exactly when and how to use flush_job_xxx() is up to the application, and is a balancing act. The processing of flush_job_xxx() is less efficient than that of submit_jo_xxx(), so calling flush_job_xxx() too often will lower the system efficiency. Conversely, calling flush_job_xxx() too rarely may result in some jobs seeing excessive latency.

There are several strategies that the application may employ for flushing. One usage model is that there is a (thread-safe) queue containing work items. One or more threads put work onto this queue, and one or more[3] processing threads remove items from this queue and process them through the MB_MGR. In this usage, a simple flushing strategy is that when the processing thread wants to do more work, but the queue is empty, it then proceeds to flush jobs until either the queue contains more work, or the MB_MGR no longer contains jobs (i.e. that flush_job_xxx() returns NULL). A variation on this is that when the work queue is empty, the processing thread

---

[3] If multiple threads are processing jobs from the same queue, then unless the application takes steps to prevent this, the jobs may be completed in a different order than that in which they entered the queue.

might pause for a short time to see if any new work appears, before it starts flushing.

In other usage models, there may be no such queue. An alternate flushing strategy is to have a separate "flush thread" hanging around. It wakes up periodically and checks to see if any work has been requested since the last time it woke up. If some period of time has gone by with no new work appearing, it would proceed to flush the MB_MGR (after taking necessary inter-thread interlocks to prevent the main thread from accessing the MB_MGR while the flush is in progress).

Job structure

At a high level, the paradigm is that the application gets an object that represents a job, where a job is a unit of work. It corresponds to one packet or buffer that needs to undergo encryption and authentication or to undergo authentication and decryption.

The job object/structure is filled in with all of the information needed to process that job. It is then returned to the system for processing. At this time a job object may or may not be returned to the application, where the returned job has completed its processing. In general the returned job, if any, will not be the same as the submitted job. However, the jobs *will* be returned in the *same order* that they were submitted.

The job structure is defined as:

```
typedef struct {
    const UINT32 *aes_enc_key_expanded;  /* 16-byte aligned pointer. */
    const UINT32 *aes_dec_key_expanded;
    UINT64 aes_key_len_in_bytes; /* Only 16, 24, and  32 byte (128, 192 and 256-
                                    bit) keys supported at this time. */
    const UINT8  *src; /* Input. May be cipher text or plaintext. In-place
                          ciphering allowed. */
    UINT8   *dst; /* Output. May be cipher text or plaintext. In-place ciphering
                     allowed, i.e. destination = source. */
    UINT64 cipher_start_src_offset_in_bytes;
    UINT64 msg_len_to_cipher_in_bytes; /* Max len = 65472 bytes. */
    UINT64 hash_start_src_offset_in_bytes;
    UINT64 msg_len_to_hash_in_bytes; /* Max len = 65496 bytes. */
    const UINT8 *iv; /* AES IV. */
    UINT64 iv_len_in_bytes; /* AES IV Len in bytes. */
    UINT8 *auth_tag_output; /* HMAC Tag output. This may point to a location in
                               the src buffer (for in place)*/
    UINT64 auth_tag_output_len_in_bytes; /* HMAC Tag output length in bytes.
                                            (May be a truncated value)*/

    /* Start algorithm-specific fields */
    union {
        struct _HMAC_specific_fields{
            const UINT8 *_hashed_auth_key_xor_ipad; /* Hashed result of HMAC key
                                                       xor'd with ipad (0x36). */
            const UINT8 *_hashed_auth_key_xor_opad; /* Hashed result of HMAC key
                                                       xor'd with opad (0x5c). */
        } HMAC;
        struct _AES_XCBC_specific_fields{
            const UINT32 *_k1_expanded;   /* 16-byte aligned pointer. */
            const UINT8  *_k2;            /* 16-byte aligned pointer. */
            const UINT8  *_k3;            /* 16-byte aligned pointer. */
        } XCBC;
    } u;

    JOB_STS  status;
    JOB_CIPHER_MODE cipher_mode; // CBC or CNTR
    JOB_CIPHER_DIRECTION cipher_direction; // Encrypt/decrypt
                                           // Ignored as the direction is implied
                                           // by the chain _order field.
    JOB_HASH_ALG hash_alg; // SHA-1 or others...
    JOB_CHAIN_ORDER chain_order; // CIPHER_HASH or HASH_CIPHER
    void    *user_data;
    void    *user_data2;
} JOB_AES_HMAC;

#define hashed_auth_key_xor_ipad u.HMAC._hashed_auth_key_xor_ipad
#define hashed_auth_key_xor_opad u.HMAC._hashed_auth_key_xor_opad
#define _k1_expanded             u.XCBC._k1_expanded
#define _k2                      u.XCBC._k2
#define _k3                      u.XCBC._k3
```

Most of the fields should be self-explanatory. The data to be encrypted or decrypted starts at (src + cipher_start_src_offset_in_bytes) and extends for a length of msg_len_to_cipher_in_bytes. The data to be hashed starts at (src + hash_start_src_offset_in_bytes) and extends for a length of msg_len_to_hash_in_bytes.

The output of the encryption/decryption is (dst). The encryption can be done "in place", i.e. (dst) can be equal to (src + cipher_start_src_offset_in_bytes).

The msg_len_to_hash_in_bytes can be any **non-zero** value. The msg_len_to_cipher_in_bytes can be any **non-zero multiple** of the cipher block size.

In the present version of the code, auth_tag_output_len_in_bytes **must be** 12. No other value is supported.

The cipher_direction field indicates whether the data should be encrypted or decrypted. The chain_order field indicates whether the crypto or hash operation should be done first. This is provided in the API in order to support possible future enhancements. However, in IPsec, the hash is always done on the cipher text rather than the plain text. So the only valid combinations of these parameters are "ENCRYPT / CIPHER_HASH" or "DECRYPT / HASH_CIPHER". Because of this, the cipher_direction field is actually ignored, and its value is inferred from the value of chain_order. However, it is always safer (to account for future changes) to set both of these values correctly.

If an invalid parameter is passed in, then when the job object is returned, it will have a status of STS_INVALID_ARGS. Otherwise, it will have a status of STS_COMPLETED. Note that in general, it will not be returned immediately if the arguments are invalid. This is because the jobs are returned in the same order in which they were submitted.

There are two "user_data" fields in the structure. These are not used by the IPsec code and can be used by the application to associate other data with the job.

Pre-expanded AES Keys

In the AES algorithms, the primary key is "expanded" into an array of keys, each of which is used for one round. To avoid having to expand the key for every buffer/packet, the API takes a pointer to an array of pre-expanded keys rather than the key itself.

The sizes of the data fields are given in the table below:

Algorithm	Key size in bytes	Expanded key array size in bytes
AES-128	16	176 = 16 * 11
AES-192	24	208 = 16 * 13
AES-256	32	240 = 16 * 15

The API to generate the expanded key values is:

```
void aes_keyexp_128_xxx(void *key,
                        void *enc_exp_keys,
                        void *dec_exp_keys);

void aes_keyexp_192_xxx(void *key,
                        void *enc_exp_keys,
                        void *dec_exp_keys);

void aes_keyexp_256_xxx(void *key,
                        void *enc_exp_keys,
                        void *dec_exp_keys);
```

where *key* points to the key, *enc_exp_keys* points to appropriately-sized buffer to receive the expanded keys for encryption, and *dec_exp_keys* points to a buffer to receive the expanded keys for decryption.

These arrays need to be 16-byte aligned for use with the IPsec APIs, so one way to declare them (using an OS-neutral alignment macro defined in os.h)would be:

```
DECLARE_ALIGNED(unsigned char enc_exp_keys[16*15], 16);
DECLARE_ALIGNED(unsigned char dec_exp_keys[16*15], 16);
```

In this way, the arrays are sized large enough to hold any of the AES expanded keys. These expanded key arrays are then passed into the IPsec APIs as inputs representing the keys.

There is also a function to expand just the encryption keys, which is needed for GCM:

```
void aes_keyexp_128_enc_xxx(void *key, void *enc_exp_keys);
```

HMAC IPad and OPad

In the HMAC algorithm, the underlying hash is performed on two buffers. Each of these buffers is pre-pended with a one-block long buffer consisting of a fixed pattern XORed with a secret key. (The details can be found in [3].)

Implemented directly, each of these blocks would have to be re-hashed for every data packet. But this is wasteful, as the same key is used for many packets. So instead of taking the secret key as input, the IPsec API takes the results of applying the underlying hash algorithm on each of these two blocks. This then becomes the starting state for hashing the rest of the data.

To assist with this process, there are a set of function to perform a raw hash of a single block:

```
void sha1_one_block_xxx(void *data, void *digest);
void sha224_one_block_xxx(void *data, void *digest);
void sha256_one_block_xxx(void *data, void *digest);
void sha384_one_block_xxx(void *data, void *digest);
void sha512_one_block_xxx(void *data, void *digest);
void md5_one_block_xxx(void *data, void *digest);
```

These functions will initialize the digest, hash a single data block, and then return the result. The digest sizes are given in the following table:

Algorithm	Digest size in bytes	Block size in bytes
MD5	16 = 4 * 4	64
SHA-1	20 = 4 * 5	64
SHA-224	32 = 4 * 8	64
SHA-256	32 = 4 * 8	64
SHA-384	64 = 8 * 8	128
SHA-512	64 = 8 * 8	128

Note that in the case of SHA-224 and SHA-384, the entire (256-bit and 512-bit respectively) digest is returned rather than the truncated digest.

The digests do not need to be aligned in particular.

For example, to compute the IPad for HMAC/SHA-1, one could use code similar to:

```
unsigned char opad[64];
for (i=0; i<64; i++) opad[i] = 0x5c;
for (i=0; i<key_size; i++) opad[i] ^= key[i];
sha1_one_block_xxx(opad, opad_hash);
```

Similar code would be used for the ipad, except for each byte of the buffer being initialized with 0x36.

AES XCBC Precomputes

The AES XCBC algorithm is defined in [4]. It defines three 16-byte keys (K1, K2, and K3) derived from the secret key. K1 is used to encrypt the data, so it needs to be expanded as described earlier. So the sizes of the three data structures are:

Field	Size in bytes
K1	176 = 11*16
K2	16
K3	16

They are generated / expanded by:

```
void aes_xcbc_expand_key_xxx(void *key,
                             void *k1_exp,
                             void *k2,
                             void *k3);
```

Selecting a Set of Functions

Some applications might want to only use the SSE functions, or the AVX functions, etc. Other applications might want to choose the family of functions at run time.

One way that this could be done is via conditional branches based on some flag. For example, this could be wrapped in a macro along the lines of:

```
#define submit_job(mb_mgr) \
   if (_use_avx2)     submit_job_avx2(mb_mgr); \
   else if (_use_avx) submit_job_avx(mb_mgr); \
   else               submit_job_sse(mb_mgr)
```

Another approach would be to embed the function addresses into a structure, call them indirectly through this structure, and change the structure based on which family should be used. For example:

```
struct funcs_t {
    init_mb_mgr_t       init_mb_mgr;
    get_next_job_t      get_next_job;
    submit_job_t        submit_job;
    get_completed_job_t get_completed_job;
    flush_job_t         flush_job;
};

funcs_t funcs_sse = {
    init_mb_mgr_sse,
    get_next_job_sse,
    submit_job_sse,
    get_completed_job_sse,
    flush_job_sse
};
funcs_t funcs_avx = {
    init_mb_mgr_avx,
    get_next_job_avx,
    submit_job_avx,
    get_completed_job_avx,
    flush_job_avx
};
...
funcs_t *funcs = &funcs_sse;
...
if (do_avx)
    funcs = &funcs_avx;
```

```
...
funcs->init_mb_mgr(&mb_mgr);
```

GCM API

Since the GCM code is implemented in an efficient single-buffer manner, there is no advantage to trying to process GCM jobs using the multi-buffer interface. Therefore, the GCM code is packaged as a separate set of single-buffer routines, which are essentially the code described in [2].

Similar to the multi-buffer interface, the GCM API comes in three families, with suffixes "_sse", "_avx_gen2" and "_avx_gen4". This reflects the implementation where the gen4 version has been optimized for generation-4 processors, but it still uses the AVX instruction set (i.e. the gen4 code will still run on gen2 processors).

Within each family, there are three functions. One takes the hash_subkey and pre-computes a number of values into a data structure. The other two perform either an encrypt or a decrypt operation using those pre-computed values.

Note also that key expansion of the primary key needs to be done before the pre-computes. So a typical sequence of operations would be:

```
gcm_data gdata;
…
aes_keyexp_128_enc_xxx(key, gdata.expanded_keys);
aesni_gcm_precomp_xxx(&gdata, hashSubKey);
…
aesni_gcm_enc_xxx(&gdata, cipher_text, plain_text, text_size,
                  iv, aad, sizeof(aad),
                  auth_tag, sizeof(auth_tag));
// or
aesni_gcm_dec_xxx(&gdata, plain_text, cipher_text, text_size,
                  iv, aad, sizeof(aad),
                  auth_tag, sizeof(auth_tag));
```

For more details on the GCM interface, see the comments in "gcm_defines.h" or [2].

Building

A Linux Makefile is provided in the release. It will build the sources into a library. That Makefile will need to be tweaked to point to the user's local version of YASM.

There is a sub-directory, LibTestApp, which has a small test application and associated trivial Makefile for verifying that all of the required APIs are in the library.

The main thing to note about building is that for Linux, for both the C and ASM files, the pre-processor symbol "LINUX" needs to be defined. For a Windows build, the symbol "WIN_ABI" needs to be defined.

There are three top-level include files:

mb_mgr.h	Main include file for Multi-buffer API
aux_funcs.h	Auxiliary functions needed to assist with Multi-buffer API
gcm_defines.h	GCM interface

A (fairly trivial) example of using these can be found in LibTestApp.

Performance

The performance results provided in this section were measured on widely available Intel® Processors. The SSE version was run on an Intel® Xeon® processor X5670, running at 2.9 GHz (SSE instruction set), and the AVX1 version was run on an Intel® Core™ i7 processor 2600, running at 3.4 GHz (AVX instruction set, second generation). In each case, the buffer size was swept in 64-byte increments. The tests were run with Intel® Turbo Boost Technology off.

Methodology

We measured the performance of the functions on data buffers of different sizes. For each size, we called the functions to process the same buffer a large number of times in a loop, to generate one timing measurement. This process was repeated a number of times, collecting many timing measurements. The main processing functions were included in the loop, but the pre-compute/auxiliary functions (e.g. AES key expansion) were not.

For each data buffer, we discarded the first and last 1/8th samples, sorted the timings, discarded the largest/smallest quarter, and averaged the remaining quarter.

The timing was measured using the **rdtsc**() function which returns the processor time stamp counter (TSC). The TSC is the number of clock cycles since the last reset. The 'TSC_initial' is the TSC recorded before the function is called. After the function is complete, the **rdtsc**() was called again to record the new cycle count 'TSC_final'. The effective cycle count for the called routine is computed using

# of cycles = (TSC_final-TSC_initial).

A large number of such measurements were made for each data buffer and then averaged as described above to get the number of cycles for that buffer size. Finally, that value was divided by the buffer size times the number of iterations of the inner loop to express the performance in cycles per byte. This was then divided into the clock rate to get the performance in bytes/second.

Note: *Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.*

For more information go to http://www.intel.com/performance

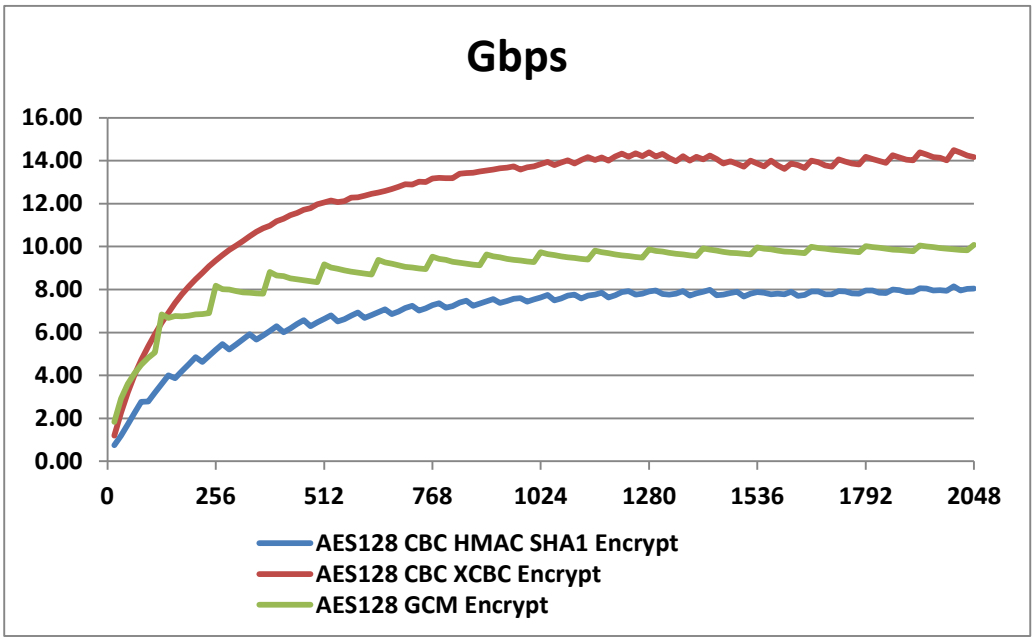
Results

There are too many combinations of algorithms to give results for each combination. So the results will be presented for three of these. These were chosen to be illustrative of the performance and to emphasize different features of the processor:

Algorithms	Emphasizes
AES128 CBC Encrypt / HMAC SHA1	Vector Registers
AES128 CBC Encrypt / XCBC	AESNI
GCM Encrypt	AESNI, PCLMULQDQ

For the HMAC and XCBC examples, the indicated buffer size is the size of the data being encrypted. The data to be hashed is 24 bytes larger, to reflect IPsec normal usage. For the GCM examples, the buffer size reflects the size of the data being encrypted *and* the data being hashed.

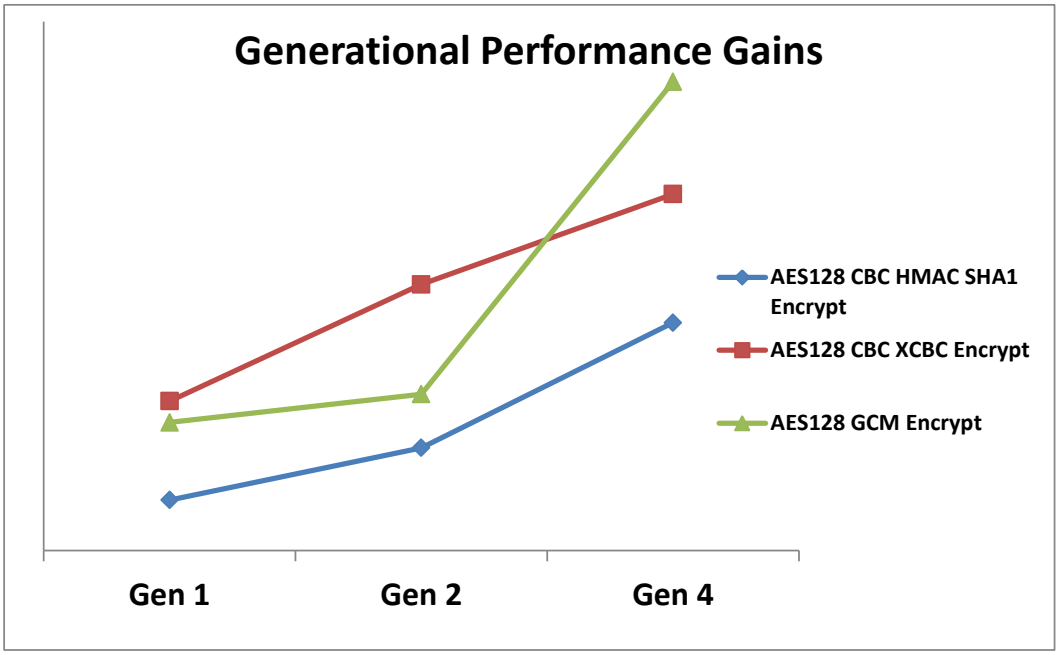
The following figure gives the single-thread throughput as a function of buffer size for the AVX version of the code (when run on an Intel® Core™ i7 processor 2600, running at 3.4 GHz)[4]:



[4] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Configurations: Refer to the Performance section on page 16. For more information go to http://www.intel.com/performance.

The following figure gives the relative performance of these algorithms for the SSE, AVX, and AVX2 versions of the code, when run on corresponding processors (all normalized to the same clock rate)[5]:



Note the dramatic increase in performance of GCM on Gen-4 due to PCLMULQDQ improvements.

Conclusion

This paper presents three IPSec implementations, optimized for different generations of Intel® processors. It describes how to build and use the library, and it presents some basic performance data.

[5] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Configurations: Refer to the Performance section on page 16. For more information go to http://www.intel.com/performance.

Contributors

We thank Malini Bhandaru and Gil Wolrich for their substantial contributions to this work.

References

[1] "Processing Multiple Buffers in Parallel to Increase Performance on Intel® Architecture Processors"

[2] "Enabling High-Performance Galois-Counter-Mode on Intel® Architecture Processors

[3] RFC2104: "HMAC: Keyed-Hashing for Message Authentication" http://tools.ietf.org/html/rfc2104

[4] RFC3566: "The AES-XCBC-MAC-96 Algorithm and Its Use With IPsec" http://www.ietf.org/rfc/rfc3566.txt

[5] Optimized IPSec Cryptographic Library: http://www.intel.com/p/en_US/embedded/hwsw/software/crc-license?id=6543

Authors

Jim Guilford, Vinodh Gopal, Sean Gulley, Erdinc Ozturk, Kirk Yap, and Wajdi Feghali are IA Architects with the IAG Group at Intel Corporation.

Acronyms

IA	Intel® Architecture