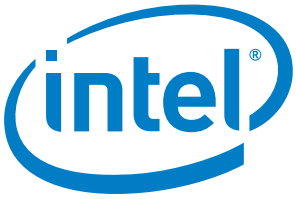


White Paper

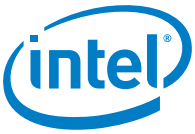
Sean Gulley
Vinodh Gopal
Jim Guilford
Kirk Yap
Wajdi Feghali
IA Architects
Intel Corporation

Multi-Hash

A Family of Cryptographic Hash Algorithm Extensions

July 2012

327739-001



Executive Summary

The paper proposes extensions to cryptographic hash algorithms, such as SHA, that add support for parallel processing of a single message. The goal is to take concepts from tree hashing and apply the parallel performance benefits to a single data buffer in a single threaded core of a modern microprocessor. Additionally, a method for applying the Multi-Hash concept to HMAC is suggested.

The paper describes the overall design of the Multi-Hash extensions, delves into details of a proposed implementation, and presents a summary of the performance of some versions of the code. With our implementation, a single core of an Intel® Core™ i7 processor 2600 with Intel® HT Technology can compute Multi-Hash SHA-256 of a 1MB buffer at the rate of ~5 cycles/byte[1] , which is over 2X faster than the best known SHA-256 single buffer implementations.

[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 of Multi-Hash section on page 9. For more information go to http://www.intel.com/performance.

Contents

Overview

Cryptographic hashes such as MD5, SHA1, SHA256 and SHA512 [1] are expensive in terms of computation on general purpose processors. They work on a single buffer of data sequentially, updating a hash digest state with the computations derived from each data block, using a number of rounds of processing that are dependent on each other. The sequential processing of the blocks of a single buffer seriously limits the performance on modern processors. Methods such as multi-buffer processing using vector Single Instruction Multiple Data (SIMD) units have been proposed [2] for better performance on usages where it is possible to work on multiple independent data buffers. However these are not applicable to usages of hashing a single buffer. Tree hashing has been around for many years [3], however, its usage has generally evolved to be thought of across multiple cores or engines. We aim to take the concepts of tree hashing and apply them to the goal of the highest possible performance on a single buffer in a single thread of a single core of a modern microprocessor.

In this paper, we describe a set of extensions to existing or future cryptographic hash algorithms that can achieve multi-buffer performance on a single buffer at comparable security strengths of the underlying hash algorithm, and generate a (different) digest of the same size as the original hash algorithm. The performance gains will be roughly proportional to the SIMD data-path width of the processor.

Some critical usages that require fast hashing of a single buffer are:

- Secure loading of files during boot/resume of a system
- Streaming applications where it is infeasible to buffer many streams for later processing

The basic idea behind Multi-Hash is to treat the single buffer as a set of interleaved independent buffers, which we call segments, and generate a number of independent hash digests for those segments in parallel. We use the notation x*i* to denote treating the data buffer as a set of i interleaved segments, where $i > 0$. It is expected that for efficiency *i* will be a power of 2. The set of digests from the parallel segments are then hashed to form a final digest of the same size as the original underlying hash algorithm. The method of interleaving the data at a fine granularity is one of the main differences of Multi-Hash and current tree-hashing implementations, which tend to break buffers down into blocks or greater.

This technique is meant to accelerate processing of hashing a single data buffer on a single thread of a single core of a modern microprocessor. However, the technique can easily be defined to be across multiple cores of a microprocessor for a second level of parallelism and performance.

Definition of Multi-Hash

We define Multi-Hash extensions formally in this section. One of the considerations in defining these extensions is how to hash a data buffer of any arbitrary length, given that the underlying hash algorithm works on blocks of a specific size (e.g. 64 Bytes). The standards that define these algorithms specify a padding scheme, whereby the buffer is typically extended with some bytes comprising a fixed pattern and the length of the buffer, to make the padded buffer the smallest multiple of the block-size.

Let us denote the hash algorithm as H, the number of parallel segments as S, the block-size as B bytes, digest-size as D bytes and width of the data words specified in the algorithm as W bytes (where B is a multiple of W).

The following proposed schemes can be considered:

1. Pad the buffer with the fixed pattern concatenated with length until the total length is a multiple of B*S. Now we can process this buffer efficiently with S-way SIMD processing, generating S digests. We treat the set of digests as another data buffer of length S*D, and then generate a final hash of size D.

2. We hash the largest region of the buffer whose length is a multiple of B*S, in parallel, generating S digests. We treat the set of digests concatenated with the rest of the buffer as a new data buffer, and then generate a final hash of size D.

3. We pad each segment to its nearest multiple of B bytes, and then process that buffer with S-way SIMD processing, generating S digests. Here the per-segment padding is done with the standard padding. One downfall with this approach is some segments may have a different padded length than other segments. We treat the set of digests as another data buffer of length S*D, and then generate a final hash of size D in a single buffer fashion.

Other variations are possible, but not as efficient as option 1. Option 1 is therefore our primary proposal for Multi-Hash.

Multi-Hash Scheme to Extend Hash Algorithm H

Consider a hash algorithm H that is defined to work on an integral number of blocks of size B bytes each. Let us denote the associated padding function as Pad_H (Message, Length of Message, Block-size B), which extends the message with a pre-determined pattern and a concatenation of the message length, to the smallest length that is a multiple of B bytes.

For a message M_0 of length L to be hashed with a given level of parallelism S, we formally define the process of Multi-Hash as follows (where the || symbol denotes concatenation):

1. Apply $Pad_H(M_0, L, \mathbf{B*S})$ to message M_0, generating M_0'of length L'. L' is the smallest length that we can extend M_0 to, that is a multiple of **B*S** bytes.

2. Divide the padded message M_0' from step 1 into S segments each of length L'/S. The padded message M_0' is divided in an interleaved fashion such that every word size W-bits of M_0' is assigned to a different segment. Each segment is represented as an array of W-bit words:

$Seg_0 = M_0'[0] \,||\, M_0'[S] \,||\, M_0'[2S] \,||\, \ldots$

$Seg_1 = M_0'[1] \,||\, M_0'[S+1] \,||\, M_0'[2S+1] \,||\, \ldots$

...

$Seg_{S-1} = M_0'[S-1] \,||\, M_0'[(2S-1)] \,||\, M_0'[(3S-1)] \,||\, \ldots$

Where each $M_0'[n]$ is a word size W index into the padded message.

3. Generate S leaf-level digests on the segments as $D_k = H(Seg_k)$ for k=0…(S-1)

4. Create a new message M_1 by interleaving the resultant digests from step 3 by every word size W-bits. Let $M_1 = D_0[0] \,||\, D_1[0] \ldots ||\, D_{(S-1)}[0] \,||\, D_1[1] \ldots ||\, D_{(S-1)}[(D/W)-1]$. Where each $D_k[n]$ is a word size W index into a segment's digest. Generate padded M_1' as $Pad_H(M_1, S*D, B)$

5. Return $H(M_1')$

For example, SHA-1 on a SIMD capable microprocessor with 128-bit registers would have the following parameter settings: B=64 Bytes, W=4 Bytes, S=4, D=20 Bytes.

Figure 1. x4 (S=4) Multi-Hash Segmentation of Message M_0

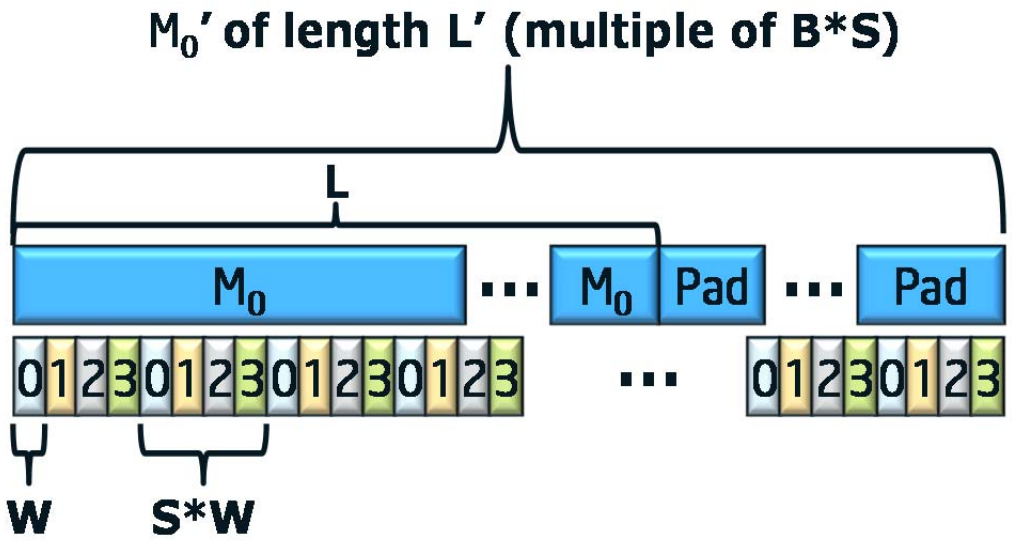
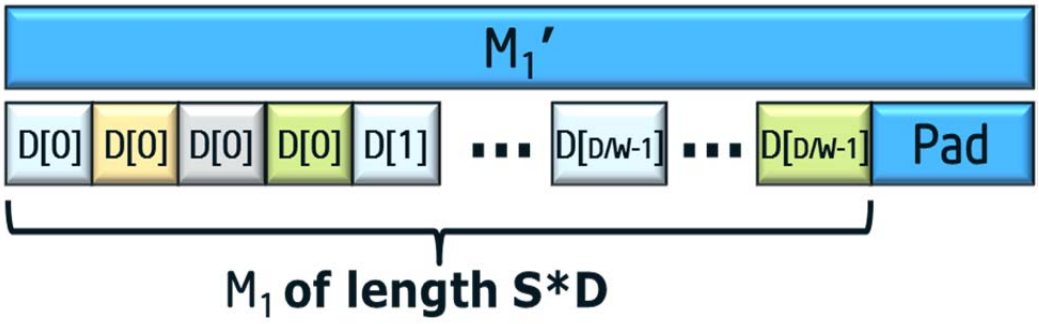


Figure 2. x4 (S=4) Multi-Hash depiction of digests with padding into M_1' for final hash



The method defined has benefits that tree hashing across buffers/cores/engines misses. For instance, the data aligned in memory is read directly into SIMD registers without the need for transposing. The method also allows data being streamed (i.e. from a network connection) to be fed directly into the Multi-Hash function without the need for knowing the length of the buffer at start time. One large advantage is that single thread applications do not have to be modified (other than at the hash algorithm level) to take advantage of greater performance due to parallelism.

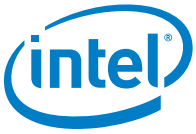
Picking a Specific Multi-Hash Function in the Family of Extensions

The new hash functions can be ordered based on compute/security considerations, and the current (possibly ordered) list of cryptographic hash algorithms in various protocols/standards can be augmented with these extensions (e.g. SHA1x4, SHA1x8, SHA256x4, SHA256x8, …).

For usages of verifying signatures of files that are securely loaded, the signing entity has to replace the current cryptographic hashing algorithm of the chosen security (e.g. SHA256), with one of the compatible extensions that is most efficient to compute for verification. For instance, if the verifying entity has a 128-bit SIMD data-path execution unit in its processor core, and if we wanted a SHA256 strength digest, it would ideally prefer SHA256x4 (as the SHA256 algorithm is 32-bit based, on a 128-bit SIMD execution unit we can process 128/32 = 4 segments, in parallel). Thus instead of using one of the currently used 32-bit algorithms {MD5, SHA1, SHA256}, the verifying entity would prefer {MD5 x8, SHA1 x4, SHA256 x4} respectively. MD5 is a bit unique in the sense that although we need only 4 segments from a 128-bit SIMD perspective, the algorithm has a very constrained data-dependency chain which makes it difficult to get the best throughput of execution units without additional parallelism.

One interesting challenge that arises is that there may be many verifying devices of different computation strengths, and the signing entity has to find the level of parallelism that works for the majority of its verifying devices. Our scheme does not require the server to estimate this very accurately, as we can always create a larger level of parallelism while signing, and have the verifying agents perform a multi-pass approach during verification if their SIMD or hardware capability cannot process as many segments as specified, all at once. For example, a signer can use a x4 scheme while a verifying agent could do two passes of a x2.

There could be some loss of efficiency if too many passes are needed, due to managing multiple state variables of the digests. Note that data can still be brought in efficiently in a streaming manner just once, however, the application will need to cycle though the sets of state variables. For instance, assume a client device does not have a SIMD unit at all, and needs to perform simple scalar operations to process a SHA256x4 hash. Instead of working with 1 set of SHA256 state variables (32 Bytes), it will simultaneously work on 4 such copies of state variables (128 Bytes), cycling through them as it processes words from the data buffer. This increase in state size is very small. A possible concern would be the working-set size increase associated with message schedules for a block (for SHA). If this increase in working-set size is problematic, one could choose to store 4 blocks of data and strictly work on one interleaved block at a time. Many other variations are possible, and we believe these methods will permit any



device to process a parallel hash signature efficiently without undue burden. The exception in these scenarios would be using a fixed hardware engine designed to perform the entire hash function, including padding, on a given buffer/length input. Unless the padding designed in the parallel Multi-Hash implementation was exactly the same as the hardware, the same result could not be calculated. However, if the hardware engine works on a per block basis or has a mode that does not include padding, then it can be used to perform multi-hash.

Based on the above considerations, we do not recommend always picking a very large number, such as x32 or x64. For instance, if most current devices are capable of x4, it would be reasonable to pick something like x8 to take advantage of the future possibility of increased SIMD data-path.

Performance of Multi-Hash

Multi-Hash provides a several factor performance gain for large message sizes. For workloads with buffers typically below 1KB, the parallelization benefits of Multi-Hash will start to diminish. Application or secure protocol designers can consider using the underlying hash function for small messages and use Multi-Hash for larger messages to achieve optimal performance.

The Multi-Hash implementation was run on an Intel® Core™ i7 processor 2600. The tests were run with Intel® Turbo Boost Technology off.

Methodology

We measured the performance of the functions on data buffers of size 1 MB. We called the functions to hash the same buffer a large number of times, collecting many timing measurements. We discarded the first and last 1/8th samples, sorted the timings, and then discarded the largest/smallest quarter, leaving the remaining quarter to be averaged.

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 and then averaged as described above to get the number of cycles. Finally, that value was divided by the buffer size to express the performance in cycles per byte.

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.

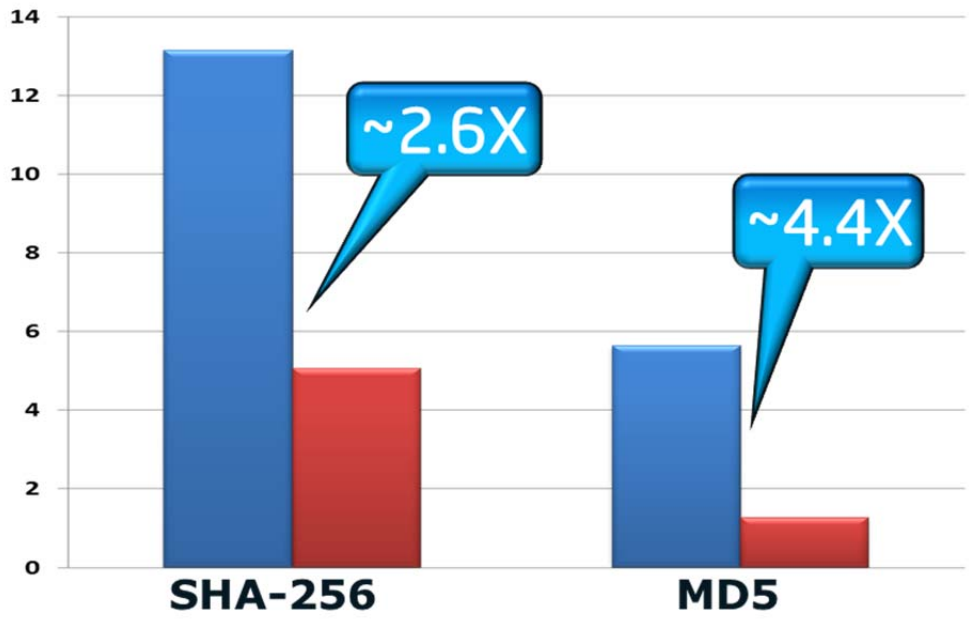
Note: For more information go to *http://www.intel.com/performance*

Results

The performance of this family of hash functions is expected to be roughly the same as a multi-buffer method. In fact, it is expected to be a little better for at least the following reasons:

- No scheduling of buffers (of possibly different lengths) in a queue
- Ideal interleaved layout that permits a streaming access of data and maps directly to SIMD execution units without requiring transpose operations on the data

Figure 3. SHA-256 and MD5 Single Buffer (Blue) vs. Multi-Hash (Red) Performance in Cycles/Byte[2]



[2] 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 of Multi-Hash section on page 9. For more information go to http://www.intel.com/performance.

For SHA256, it can be seen that the SHA256x4 Multi-Hash provides ~2.6X performance gain over the best SHA256 algorithm computation on a reasonably sized 1MB data buffer [4].

For MD5, the ideal parallelism is actually larger than the SIMD width due to the latencies of execution and tight data dependency chain of the round function. Therefore MD5 shows ~4.4X[3] performance gain with Multi-Hash (utilizing a Multi-Hash x8 as opposed to a Multi-Hash x4 that would be ideal for SHA1/SHA256).

It is interesting to observe that even with a 128-bit SIMD, SHA256 Multi-Hash performance is better than the single-buffer performance of the fastest still widely-used cryptographic hash algorithm MD5. We expect Multi-Hash performance to scale proportional to increasing SIMD data-path widths of future processors.

Security of Multi-Hash

It can be seen that our proposed method is a special case of tree-mode hashing, where, at the leaf-level we hash interleaved words of the data buffer, and in a 2nd level we hash the ordered set of hash digests. We perform padding at both levels of the tree, derived from the standard for the underlying hash function. The fan-out of the tree is S. We hypothesize that the resulting digest from Multi-Hash is at least as secure and collision-resistant as the digest obtained by a direct application of the underlying hash function.

SHA-3 Candidates

In addition to the most commonly used hash functions today, Multi-Hash extensions should work for the new SHA3 candidates as well.

The definition of Multi-Hash aligns closely with the tree hashing parameters outlined by the Keccak team in [5]. In [6], the Keccak team describes the tree hashing parameter values for a 128-bit data path SIMD implementation of Keccak as (G=LI, H=1, D=2, B=64, C=c=576). Trying to keep within a similar framework for Multi-Hash, the D value is equal to S, the number of segments. G would always be equal to LI, H is 1, and B would be equal to

[3] 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 of Multi-Hash section on page 9. For more information go to http://www.intel.com/performance.

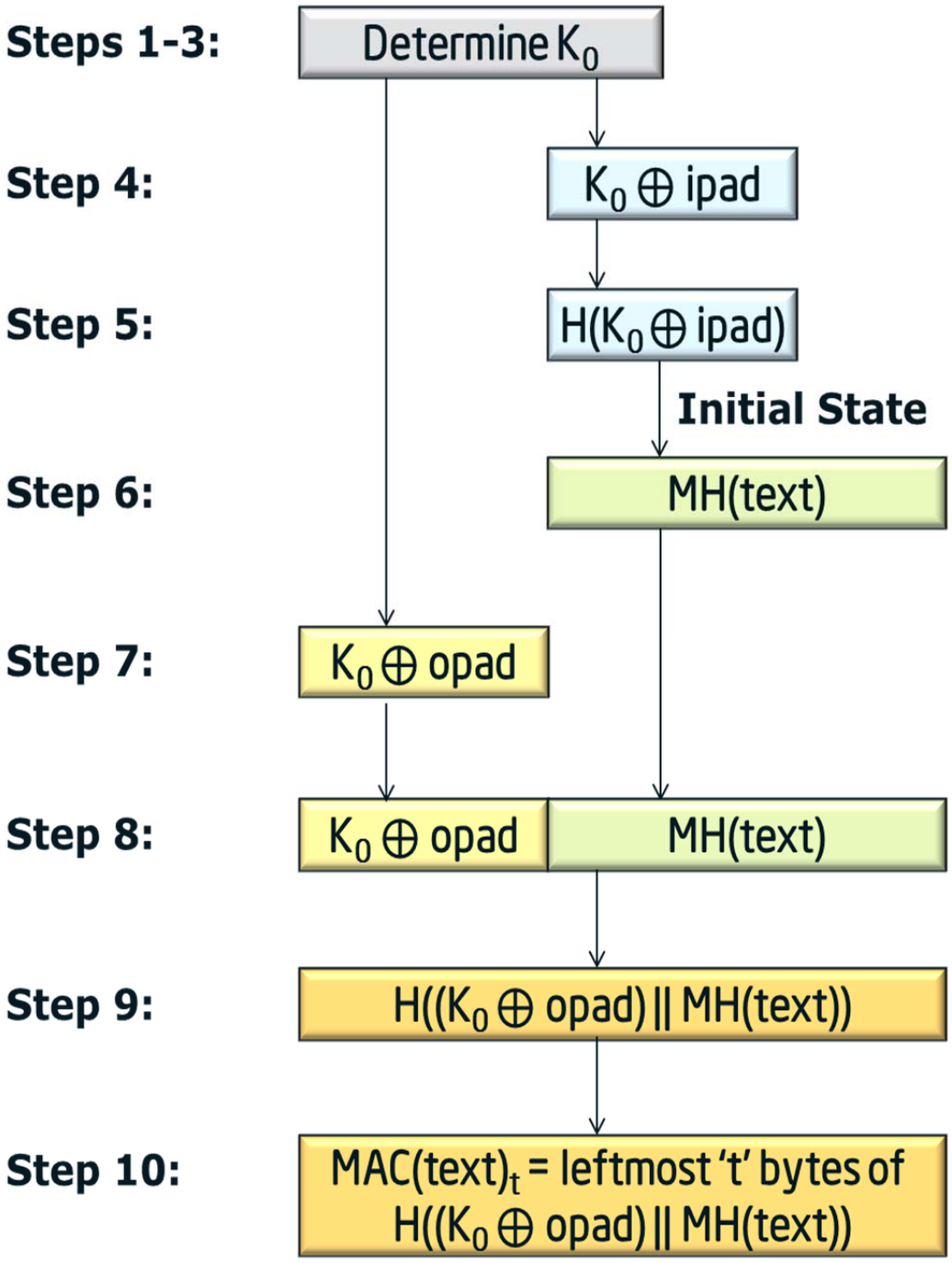
the word size in bits (e.g. 32 for SHA1/2). The chaining value, C, would be equal to digest size in bits (e.g. 256 for SHA256).

Multi-Hash and HMAC

Hash-based message authentication code (HMAC) is a mechanism for creating a message authentication code (MAC) using a cryptographic hash function with the combination of a message and secret key [7]. Secure network connections, using protocols such as IPSec and TLS, make use of HMACs to ensure a message has been received correctly from the expected sender. In scenarios where messages may be large, performance may be improved if the HMAC is calculated in a parallel manner. Using the notation from the FIPS-198 specification, with the addition of parameter *MH* being an approved Multi-Hash function, we define the Multi-Hash HMAC function as follows:

1. If the length of $K=B$: set $K_0 = K$. Go to step 4.
2. If the length of $K > B$: hash K to obtain an L byte string, then append $(B-L)$ zeros to create a B-byte string K_0 (i.e., K_0 = H(K) || (00..00). Go to step 4.
3. If the length of $K<B$: append zeros to the end of K to create a B-byte string K_0 (e.g., if K is 20 bytes in length and B=64, then K will be appended with 44 zero bytes 0x00).
4. Exclusive-Or K_0 with *ipad* to produce B-byte string: $K_0 \oplus ipad$.
5. Apply H to the string generated in step 4: $H(K_0 \oplus ipad)$.
6. Use the final state of step 5 as the initial state for each segment in the application of *MH* to the stream of data '*text*': MH(*text*) (e.g., the initial state variables of the underlying hash used by each segment within the Multi-Hash are set to the final state of $H(K_0 \oplus ipad)$ as opposed to using the published initial constants). Note the initial state for step 6 may not be the actual hash result of step 5 (e.g. SHA-224, use all 256 bits of state).
7. Exclusive-Or K_0 with *opad* to produce B-byte string: $K_0 \oplus opad$.
8. Append the result from step 6 to step 7: $(K_0 \oplus opad)$|| MH(*text*)
9. Apply H to the result from step 8: H($(K_0 \oplus opad)$|| MH(*text*))
10. Select the leftmost t bytes of the result of step 9 as the MAC

Figure 4. HMAC Construction with Multi-Hash



In step 6, we can define *MH* in two different ways. We can use *MH* as the two step process that appends a single digest to the outer portion in step 8, or we could use *MH* as a one step process and append the *S* digests to the

outer portion. We propose using the latter option of appending all *S* digests to $K_0 \oplus opad$ in step 8.

Another alternative in the derivation of K0 is to divide the key into S chunks, and perform S different hashes in step 4 to provide a different initial state to each of the S segments in the MH of step 6.

Conclusion

We propose a family of extensions of the widely used cryptographic hashes, called Multi-Hash, which can greatly accelerate the hashing of a single data buffer through parallel processing. The usage of the Multi-Hash family of algorithms is expected to be very beneficial in the case where we have a single core (single thread of execution) processing data buffers one at a time. We expect the security/collision properties of the scheme to be comparable to that of the underlying hash function. Though we generate a digest of the same size/strength as the underlying hash function, we expect to generate a different one, and therefore require a new standard for Multi-Hash extensions that can then be adopted into higher-level protocols/standards. Performance gains of over 4X[4] are possible on current processors that offer 128-bit SIMD execution units, and we expect performance to scale roughly proportional to the SIMD data-path width of future processors.

Acknowledgements

We thank our Intel colleagues, David Cote, Erdinc Ozturk and Gil Wolrich for their substantial contributions to this work. We acknowledge the NIST Cryptographic Technology Group for their invaluable comments.

[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 of Multi-Hash section on page 9. For more information go to http://www.intel.com/performance.

References

[1] FIPS Pub 180-2 Secure Hash Standard
http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[2] V. Gopal, J. Guilford, W. Feghali, E. Ozturk, G. Wolrich, M. Dixon, *Processing Multiple Buffers in Parallel to Increase Performance on Intel® Architecture Processors*, July 2010, http://download.intel.com/design/intarch/papers/324101.pdf

[3] R. Merkle, *Secrecy, Authentication, and Public Key Systems*, June 1979, http://www.merkle.com/papers/Thesis1979.pdf

[4] J. Guilford, K. Yap, V. Gopal, *Fast SHA-256 Implementations on Intel Architecture Processors*, May 2012, http://download.intel.com/embedded/processor/whitepaper/327457.pdf

[5] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *Cryptographic sponge functions*, January 2011, http://sponge.noekeon.org/

[6] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *Keccak implementation overview*, January 2011, http://keccak.noekeon.org/

[7] FIPS Pub 198 The Keyed-Hash Message Authentication Code (HMAC) - http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf

Authors

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

Acronyms

IA Intel® Architecture

§