AVON
A Fast Hash Function for Intel SIMD Architectures

Matt Henricksen[1] and Shinsaku Kiyomoto[2]
[1]*Institute for Infocomm Research, 1 Fusionopolis Way #21-01 Connexis (South Tower), 138632 Singapore, Singapore*
[2]*KDDI R & D Laboratories Inc, 2-1-15 Ohara, Fujimino-shi, 356-8502 Saitama, Japan*

Keywords: Hash Functions, SHA-3, Single-instruction Multiple-data.

Abstract: In this paper, we propose a hash function that takes advantage of the AES-NI and other Single-Instruction Multiple-Data operations on Intel x64 platforms to generate digests very efficiently. It is suitable for applications in which a server needs to securely hash electronic documents at a rate of several cycles/byte. This makes it much more efficient for certain applications than SHA-2, SHA-3 or any of the SHA-3 finalists. On the common Sandy Bridge micro-architecture, our hash function, AVON, has a throughput of 2.65 cycles per byte while retaining a high degree of security.

1 INTRODUCTION

Following the successful analysis of MD-5 and SHA-1 (Wang and Yu, 2005)(Wang et al., 2005), the academic community, and to an extent, industry have been looking for replacement hash functions. In 2007, NIST launched the SHA-3 hash function competition (National Institute of Standards and Technology, 2007), as a multiple-phase effort to find suitable hash functions to complement or replace the standardized SHA-2, which seemed at the time to be vulnerable to the same set of attacks.

Of the 51 submissions, five were chosen as finalists. These were Blake, Grostl, JH, Keccak and Skein. In 2012, Keccak (Bertoni et al., 2011) was selected as the winner. There has been criticism that these finalist hash functions are too slow to meet industry needs (see Table 1, which provides median throughputs for long messages on different Intel microarchitectures (ECRYPT, 2012)). Some calls have been made for hash functions with a throughput of 2-5 cycles/bytes (Gligoroski, 2010).

One of the reasons for the conservative speed of the SHA-3 finalists is that in order to be competitive in processes such as the NIST SHA-3 hash function competition, the hash functions must offer many properties, above and beyond the well known trinity of pre-image, second pre-image and collision resistance. The hash functions must also perform well on a large number of platforms.

This means that these hash functions are not well suited to niche applications. The industry request that inspired this research identified an application in which a server rapidly hashes very many electronic documents. In this application, although the clients may use a diverse range of processors, there is no reason why the servers cannot be a specific type of processor, for which the hash function is targeted and optimized. The hash function should be very fast on the server, with throughput around the cited two cycles/byte. After the documents and their digests are distributed, each client can verify the digests on some subset of documents without the same severe time/throughput constraint.

Table 1: Throughput of the SHA-3 finalists (and SHA-2) in generating 512-bit digests for long messages (cycles/byte).

Hash Function	Westmere	Sandy Bridge	Atom
BLAKE	8.06	5.66	12.80
Grostl	30.32	26.05	110.2
JH	15.56	13.85	30.28
Keccak	12.14	10.86	21.43
Skein	7.27	6.38	9.96
SHA-2-512	12.50	11.67	15.41

In this paper, we investigate how to construct a hash function that offers pre-image and second-preimage resistance, but not necessarily collision-resistance, on Intel x86-64 architectures using Single-Instruction Multiple-Data (SIMD) sets with SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions). This processor family is very popular. The hash function's performance require-

Henricksen M. and Kiyomoto S..
AVON - A Fast Hash Function for Intel SIMD Architectures.
DOI: 10.5220/0004502104820487
In *Proceedings of the 10th International Conference on Security and Cryptography* (SECRYPT-2013), pages 482-487
ISBN: 978-989-8565-73-0

ments are asymmetric: it must perform very efficiently on the Intel x86 server using SIMD operation, but on the client, throughput is not critical. Pre-image and second pre-image resistance are required, since a rogue client might try to modify a received document without altering the digest.

In Section 2, we describe SSE/AVX in terms of relevant instructions. In Section 3 of this paper, we present the new hash function AVON, built using the well-known sponge construction, with a permutation specially designed to use SIMD instructions. In Section 4, we outline our design decisions. In Sections 5 and 6, we analyse the security and efficiency of the cipher. We provide concluding notes in Section 7.

2 PRELIMINARIES - STREAMING SIMD EXTENSIONS

Recent Intel architectures offer SIMD functionality, through the SSE and the newer AVX instruction sets. SSE instructions operate on dedicated 128-bit XMM registers, and AVX instructions operate on 256-bit YMM registers. This is in contrast to general purpose instructions, which are restricted to 32- or 64-bit registers. On recent architectures, such as Westmere, Sandy Bridge and Ivy Bridge, SSE and AVX instructions are implemented efficiently so that for many applications, increases in throughput result by using these larger register sets.

As an example of the increased power offered by SSE/AVX, consider the instruction PBLENDW, which takes two 128-bit operands and an immediate value. The operands are notionally divided into eight sixteen-bit blocks. The instruction creates a target word using blocks chosen from either of the operands, according to the bit pattern of the immediate value. This instruction can be synthesized in 32-bit C code by masking each of the source words in 32-bit blocks, and exclusive-oring the blocks together, which takes at least three cycles on general purpose registers, but only one cycle when using PBLENDW.

Of particular interest to cryptographers are the AES-NI instructions, which enable very fast implementation of the Advanced Encryption Standard block cipher and variants. The foremost instruction is the AESENC instruction, which performs one round of the AES, incorporating the non-linear SubByte (*SB*) operation (comprising sixteen parallel invocations of an optimal 8×8 s-box), and the ShiftRows (*SR*), MixColumns (*MC*) and AddKey (*AK*) operations. On Westmere, the instruction is executed with a throughput of six cycles and a latency of two cycles. On Sandy Bridge, the instruction is executed with a throughput of eight cycles and a latency of one cycle (Agner, 2012). So, for independent operands, Sandy Bridge is more efficient, but for a sequence of iterated rounds, Westmere is more efficient.

AESKEYGENASSIST is also part of the AES-NI, although its use is less straight-forward than AESENC. The AES round key schedule needs to provide 128-bit round keys for each round of AES plus one additional 128-bit key for pre-whitening. It does this in different ways for 128-, 192- and 256-bit master keys. AESKEYGENASSIST operates in such a way that it simultaneously provides support for all these master-key lengths by computing s-boxes and other operations on two of the four 32-bit words in the source operand. This provides more non-linearity than required by the weak AES key schedule, so significant manipulation is required to derive the AES keys. If we use the instruction directly, rather than manipulating it to derive the keys, we are actually benefited by the additional non-linearity it provides as well as processing more quickly than if deriving the keys. The form of the instruction is *AESKEYGENASSIST target, source, imm*, where imm is a hard-wired round constant.

3 SPECIFICATION

The AVON hash function is based on the sponge function construction to provide an easy trade-off between security and efficiency, by altering the rate at which message material is added to, or digest material is leaked from the state. The sponge function construction also provides certain guarantees about security, if its core permutation is considered ideal, simplifying analysis. In this section, we show how to implement AVON by using the sponge function with the core permutation. We also provide the details of the permutation.

3.1 Construction

The sponge function, as defined by Bertoni, Daemen, Peeters and Assche (Bertoni et al., 2007), shown in Figure 1, is a generalization of a hash function (with variable input length and fixed output length) and a stream cipher (with fixed input length and variable output length). It iteratively applies a (bijective) permutation to an t-bit state, which is divided between a capacity component of size c and a rate component of size r. The permutation does not require an inverse or a key schedule.

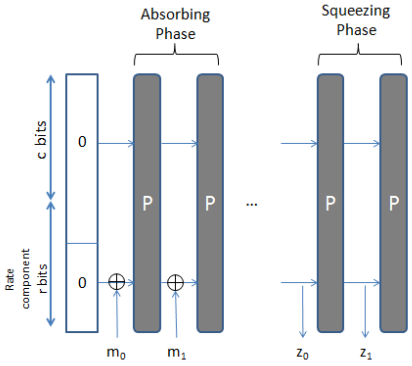

Figure 1: The sponge construction.

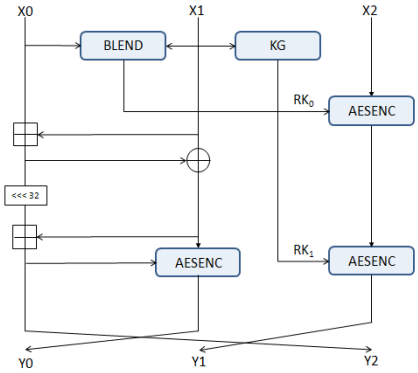

Figure 2: *Subround* - one third of the permutation.

Generating a digest requires initializing the state with a fixed constant (for example, the all-zero string), then running absorbing and squeezing phases. During the absorbing phase, message material is iteratively added to the state in r- bit blocks (m_i). After each block is absorbed, the permutation P is called to mix the state. The absorbing state is followed by the squeezing state, in which r'- bit blocks (z_i) are emitted from the state in between iterations of the permutation. These blocks are concatenated to form the digest. We assume in this article that $r = r'$.

AVON uses an internal state of 384 bits (divided between three double quadwords of 128 bits each, termed X_0, X_1 and X_2). The initial state is set to the all-zero string. The rate for absorbing and squeezing is $r = 128$. The capacity is set to $c = 256$.

During the absorbing phase, the message is broken into blocks of 128 bits. The last block is padded according to the rules given in Section 3.3. Each block is absorbed into the sponge state, by combination with the 128-bit rate component X_2, using binary addition. After each combination, the permutation P (Section 3.2) is executed.

Once the absorbing phase has finished, the squeezing mode commences to produce the digest. This is formed by extracting the 128 bits of state that is stored in the rate component X_2, invoking the permutation, then iterating. The n-bit digest is formed by concatenating the x blocks of extracted state.

3.2 Permutation

The permutation P operates on a 384-bit state $X0||X1||X2$, where each variable X_i is a 128-bit double quadword, and $||$ represents concatenation. The permutation consists of iterating the *SubRound* function three times. The *SubRound* function is shown in Figure 2.

AESENC encrypts its input P with one round of the AES using the round key RK, as documented in (Daemen and Rijmen, 2002). The round keys for the AESENC operations in the X_2 sub-state are defined as:

$$\begin{aligned} RK_0 &= (X_0 \& \text{mask}_{0,r}) \oplus (X_1 \& \text{mask}_{1,r}) \\ RK_1 &= \text{KG}(X1, c_r) \end{aligned}$$

The masks vary according to round number $0 \leq r \leq 2$. The series $\text{mask}_{1,r}$ are the inverse of $\text{mask}_{0,r}$.

$$\begin{aligned} \text{mask}_{0,0} &= \{0xFFFFFFFF, 0x00000000, \\ &\quad 0xFFFF0000, 0xFFFF0000\} \\ \text{mask}_{0,1} &= \{0xFFFF0000, 0xFFFF0000, \\ &\quad 0xFFFFFFFF, 0x00000000\} \\ \text{mask}_{0,2} &= \{0xFFFFFFFF, 0x00000000, \\ &\quad 0xFFFF0000, 0xFFFF0000\} \end{aligned}$$

The operation to derive RK_0 can be implemented using PBLENDW, with constants $0xCA$ for sub-rounds 0 and 2, and $0xAC$ for sub-round 1. The KG operation can be implemented using AESKEYGENASSIST with constants $0x1$, $0x3$ and $0x9$ for $r = 0, 1, 2$ consecutively.

The permutation P is implemented as:

$$\begin{aligned} Y0 &= \text{AESENC}((X0 + X1) \oplus X1, Y2) \\ Y1 &= \text{AESENC}(\text{AESENC}(X_2, RK_0), RK_1) \\ Y2 &= ((X0 + X1) \lll_{128} 32) + ((X0 + X1) \oplus X1) \end{aligned}$$

At the end of the sub-round, the sponge state is updated as $(X0 = Y0, X1 = Y1, X2 = Y2)$.

3.3 Padding

We use the simplest sponge compliant padding rule pad10*, which consists of padding the last block of the message with a 1 bit, followed by as many 0-bits as required to ensure that the padded message length is a multiple of the block size, eg. $n \times r$ for some integer n.

4 DESIGN PRINCIPLES

4.1 Sponge Construction

We use the sponge construction because it is the most scalable construction to trade-off efficiency and security, simply by varying the rate r. It also offers some proofs of security, given a randomized permutation. In particular, resistance against collisions, pre-images and second-preimages can be computed as (Guo et al., 2011):

- Collision: $\min(2^{\frac{n}{2}}, 2^{\frac{c}{2}})$
- Second Preimage: $\min(2^{n}, 2^{\frac{c}{2}})$
- Preimage: $\min(2^{\min(n,t)}, \max(2^{\min(n,t)-r}, 2^{\frac{c}{2}}))$

We set the size of the construction ($t = c + r$) to 384 bits so that we only need to use three registers to store the state, with a few other registers for temporary values (in 32-bit mode, there are only eight XMM registers available). Setting the capacity (c) to 256 bits means that if there are no flaws in the permutation, then for generating a 256-bit digest ($n = 256$), collision resistance = 2^{128}, second-preimage resistance is 2^{128}, and pre-image resistance is 2^{128}.

4.2 The Permutation

The permutation P is designed to meet the following requirements: P must make use of AESENC and SSE instructions; every output byte of P depends upon every input byte; P must provide similar diffusion and confusion to one full AES encryption; using SSE, the permutation must complete within 48 cycles and without SSE, it must complete within 256 cycles.

The permutation is a combination of ARX design and components of the AES. This combines both the well-established properties and efficiency of the AES, with some uniqueness in its design permitted by the SSE instruction set. Choosing the permutation to be a combination of AES and ARX should be more resilient than either in isolation.

ARX designs tend to be popular because addition is generally faster than an s-box lookup (addition is an atomic operation, an s-box lookup is a sequence of instructions). However, access to the AES-NI changes this, since an s-box lookup can be performed in an average of less than $\frac{1}{2}$ cycle. There is no way to use any s-box other than that of the AES in a competitive manner on the Intel x86-64, and due to its optimal security properties, nor is there any need.

The permutation is designed to exploit SIMD architectures. SIMD instructions are designed for specialized operations such as graphics acceleration, so is more limited than for those of the general purpose registers. For example, Intel SSE instructions are incapable of generic indirect addressing, such as used in s-boxes implemented using lookup tables.

However, AES-NI provides a single s-box implemented on chip, and accessible through SIMD. The s-box, used by the AES, is optimal against differential and linear cryptanalysis for that table size. The relevant instruction, AESENC computes sixteen s-boxes in parallel with a throughput of six to eight cycles. So it does not make sense to implement any other s-box. The AES-NI instruction set also provides AESKEYGENASSIST, which partially computes round keys. It provides more non-linearity than is used in the key generation for round keys (eight s-boxes per 128 bits rather than four). This should be exploited rather than discarded.

We also use a number of other instructions, including PBLENDW, for computing the round keys for both AESENC operations per SubRound. PBLENDW completes in one cycle, so is ideal for computing the first round key. AESKEYGENASSIST can start to compute the second round key at the same time, since its operands are not modified by PBLENDW. So the round keys are computed respectively one cycle, and 6-8 cycles after the start of the SubRound. As two AESENC operate in serial, the second round key is not required until at least six cycles after the start of the round.

5 ANALYSIS

AVON uses the sponge construction, and it is proved in (Bertoni et al., 2008) that if the underlying permutation is ideal, then the hash function constructed using a sponge function is indifferentiable from a random oracle. This means that we only need to concentrate on the security of the permutation. A more full analysis will be presented in an extended version of this paper.

5.1 Statistical Analysis

We performed statistical analysis on AVON to measure the diffusion and confusion of the permutation for 2^{27} pairs of inputs with one-byte, two-byte or random differences. 98% of samples produced at least 47 active bytes over the permutation P, showing that three rounds of *SubRound* is sufficient for diffusion purposes. All samples produced at least 35 active s-boxes, and 99% of samples with random differences produced 166, 167 or 168 (the maximum) active s-boxes. This means that even if the attacker has direct

access to the input of P, which is not possible when using the sponge function construction, he is unable to control the differences in order to produce a collision in a 256-bit digest that is more efficient than a generic attack. This is because the maximum probability of an input difference passing through an AES s-box to a specific output difference is 2^{-6}, and passing through 35 active s-boxes is 2^{-210}.

5.2 Cryptanalysis

5.2.1 Differential and Linear Cryptanalysis

The security of AVON against differential and linear cryptanalysis rests on the security of the AES, which has been well studied. The maximum probability of any non-zero input difference passing through the AES s-box to any specific output difference is 2^{-6}, and the maximum bias of the s-box is 2^{-3}. Since P activates at least 35 active s-boxes, it seems unlikely that any differential path with a probability of more than 2^{-210}, and any linear approximation with bias of less than 2^{-105} is unlikely, and the attacker will be unable to generate a collision.

5.2.2 ARX Cryptanalysis and Rotational Distinguishers

Rotational distinguishers are applied to ARX ciphers in (Khovratovich and Nikolic, 2010). The aim is to analyse how the difference between a pair of inputs, where one is a rotated version of another, passes through the components of an ARX cipher. Only the addition component of the cipher alters the difference, and the number of additions required to effect n-bit security can be simply quantified.

AVON-2 uses an ARX component which can be analysed in isolation by the rotational cryptanalysis method. However, the non-linear s-boxes in the AES component are not readily analyzable, and also provide higher non-linearity than the equivalent additions. So rotational distinguishers are unlikely to apply to AVON-2. The attacker might try to build a rotational distinguisher by carefully controlling how difference propagate through the s-boxes, either by fixing the difference or fixing values. But the large number of activated s-boxes means this approach is very unlikely to succeed.

5.2.3 Rebound Attacks

Rebound attacks (Mendel et al., 2009) combine meet-in-the-middle attacks with truncated differentials, to allow the attacker to efficiently control an intermediate segment of the hash function rounds, so that the input differential of the segment and the output differential converge to the same pattern midway. The attacker aims to reduce the number of active s-boxes subsequent to, and preceding the segment, allowing the attack to be probabilistically extended in the forward and back directions over a further number of rounds.

Rebound attacks assume full control over the intermediate values of the round function, and are difficult to convert to full hash functions that use the sponge function construction. Rebound attacks are known to be ineffective in the presence of ARX components, so seem unlikely to apply AVON.

5.2.4 Algebraic Attacks and Cube Testers

Cube attacks and cube testers are described in (Aumasson et al., 2009). To date there has been no meaningful progress in applying algebraic attacks to the AES. Since each message block is modified by 162 s-boxes, it is infeasible to apply this attack to AVON.

6 IMPLEMENTATION

The *SubRound* core is iterated three times, with different constants, to comprise P. The first instance of *SubRound* is shown, using SSE assembly, here:

```
#define SUBROUND1 \
  vpblendw $0xCA,  %%xmm1, %%xmm0, %%xmm3 \
  vaeskeygenassist $1,     %%xmm1, %%xmm5 \
  vaesenc          %%xmm3, %%xmm2, %%xmm4 \
  vpaddw           %%xmm1, %%xmm0, %%xmm0 \
  vpxor            %%xmm1, %%xmm0, %%xmm1 \
  vpshufd          $0x93,  %%xmm0, %%xmm0 \
  vpaddw           %%xmm1, %%xmm0, %%xmm0 \
  vaesenc          %%xmm5, %%xmm4, %%xmm2 \
  vaesenc          %%xmm0, %%xmm1, %%xmm1 \
```

The throughput of the cipher is limited by the serial nature of the AESENC operations on the $X2$ word.

This cipher was implemented on an Intel Core i7 Extreme 990X running at 3470 MHz based on the Westmere microarchitecture, and on an Intel Core i7 2600 running at 3400 MHz based on the Sandy Bridge architecture. The permutation P was timed at 2.65 cycles/bytes when coded on the latter machine using AVX instructions, and at 3.36 cycles/byte when coded on the former without.

AVX instructions, which operate on 256 bit registers, are not currently very useful, since for the cryptographic operations and many others, the leftmost 128 bits are zero-filled, meaning that the instructions remain oriented on 128 bits. However, AVX instructions frequently take one additional operand

Table 2: Throughput of the AVON hash function (cycles/byte).

Hash Function	Westmere	Sandy Bridge
AVON Permutation	3.27	2.58
AVON Full	3.36	2.65
MD5	5.04	5.38
SHA-1	7.66	7.80
Keccak	12.14	10.86
SHA-2-512	12.50	11.67

compared to SSE instructions, specifying a target, whereas the comparable SSE instructions overwrite one of the source operands. This means that some speedup can be obtained by using AVX, since for where the source needs to be preserved, one move operation can be saved. We note that if AVX and SSE instructions are mixed, there is a penalty applied to the throughput. The real benefit of the Sandy Bridge architecture lies in the ability to execute AESENC operations concurrently on two execution ports (on Westmere, only one port can handle this instruction).

7 CONCLUSIONS

In this document, we specify a new hash function - AVON - that specifically uses the SIMD instruction set on the Intel platform to increase throughput. We achieved a speed of 2.65 cycles/byte on the Sandy Bridge micro-architecture, which is roughly four times faster than the standardized SHA-3 on the same platform, and roughly double the speed of the insecure MD5 hash function on Sandy Bridge (see Table 2). On other platforms, it is not so effective, but this is unimportant for two reasons. Firstly, wherever AES is efficient, AVON will also be efficient. There is a strong trend to SIMD, and AES will be available as atomic SIMD instructions on more platforms in the future. Secondly, the hash function is useful in a server-client scenario, where the server has to hash many documents, and the client to verify only a few.

Future work includes optimization of the code, and further cryptanalysis to more precisely determine the security of the hash function.

REFERENCES

Agner (2012). The microarchitecture of Intel, AMD and VIA CPUs. http://www.agner.org/optimize/microarchitecture.pdf.

Aumasson, J.-P., Dinur, I., Meier, W., and Shamir, A. (2009). Cube Testers and Key Recovery Attacks on Reduced-Round MD6 and Trivium. In (Dunkelman, 2009), pages 1–22.

Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. (2007). Sponge functions. In *Proceedings of ECRYPT Hash Workshop 2007*, May 24 - 25, 2007, Barcelona, Spain.

Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. (2008). On the Indifferentiability of the Sponge Construction. In Smart, N. P., editor, *EUROCRYPT*, volume 4965 of *LNCS*, pages 181–197. Springer.

Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. (2011). The KECCAK SHA-3 Submission. Submission to NIST (Round 3). Available at http://keccak.noekeon.org/Keccak-submission-3.pdf.

Daemen, J. and Rijmen, V. (2002). *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer.

Dunkelman, O., editor (2009). *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, volume 5665 of *LNCS*. Springer.

ECRYPT (2012). ebacs: Ecrypt benchmarking of cryptographic systems. http://bench.cr.yp.to/results-sha3.html.

Gligoroski, D. (2010). Cryptographic hash functions. http://www.nisnet.no/filer/Finse10/Cryptographic_ash_Gligoroski.pdf.

Guo, J., Peyrin, T., and Poschmann, A. (2011). The PHOTON family of lightweight hash functions. In Rogaway, P., editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 222–239. Springer.

Khovratovich, D. and Nikolic, I. (2010). Rotational cryptanalysis of ARX. In Hong, S. and Iwata, T., editors, *FSE*, volume 6147 of *Lecture Notes in Computer Science*, pages 333–346. Springer.

Mendel, F., Rechberger, C., Schläffer, M., and Thomsen, S. S. (2009). The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In (Dunkelman, 2009), pages 260–276.

National Institute of Standards and Technology (2007). Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. *Federal Register*, 27(212):62212–62220. Available at http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf.

Wang, X., Yin, Y. L., and Yu, H. (2005). Finding collisions in the full SHA-1. In Shoup, V., editor, *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer.

Wang, X. and Yu, H. (2005). How to break MD5 and other hash functions. In Cramer, R., editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer.