A New Universal Hash Function: Its Integration with Block and Stream Ciphers for Encryption and Authentication

Sebati Ghosh
(CS1320)
under the supervision of
Prof. Palash Sarkar

Indian Statistical Institute, Kolkata

Abstract. In this dissertation, we designed a new two-layered keyed hash function having provably low collision and differential probabilities. The first layer of the hash function uses BRW [3] polynomial based Hash function for each portion of a fixed length of the input message and the second layer uses usual Horner's rule based Hash function on the digest stream obtained from the first layer. This hash function is then integrated with the block cipher Advanced Encryption Standard (AES) and the stream cipher Salsa20 to implement some of the encryption and authentication primitives mentioned in [1].

1 Introduction

The study in the area of symmetric key cryptography can be broadly classified along two major branches, namely Encryption and Authentication. Besides studying these in isolation, Authenticated Encryption (AE) is also studied. Block ciphers, Stream ciphers, Hash function are some of the basic primitives for performing these cryptographic operations. In [1] (to be referred to as main paper in subsequent discussion) a systematic framework for using a stream cipher supporting an initialization vector (IV) to obtain secure primitives for message authentication code (MAC), authenticated encryption (AE), authenticated encryption with associated data (AEAD) and deterministic authenticated encryption (DAE) with associated data (DAE(AD)) has been proposed. Several schemes have been presented and analyzed. Though the main paper uses only Stream ciphers supporting an IV, in this work the Block cipher AES in counter mode with an encrypted IV has been used in place of it to implement the primitives. In parallel some of the schemes have also been implemented using Salsa20 of Bernstein [2] in place of the stream cipher. Along with stream cipher, a major component of the constructions is a vector input (double-input hash function is required for AEAD schemes, the inputs being data and header part) keyed hash function having provably low collision and differential probabilities. Hence a suitable hash function has been designed and implemented in the present work satisfying the required criteria. In fact, this hash function design and implementations forms the major and most significant part of this present work.

Previous Related Works: Universal hash functions have been extensively used in cryptography, especially in the construction of message authentication code (MAC) algorithms, since these were introduced by Carter and Wegman [10]. A well-known universal hash function is the so called multi-linear universal hash function, defined as follows:
The message $M = (M_1, \cdots, M_l)$ and key $K = (K_1, \cdots, K_l)$ are sequences of elements over

a finite field $\mathbb{F}$. The multi-linear hash function is the multi-linear map:

$$\mathsf{MLHash} : (M_1, \cdots, M_l) \xrightarrow{K} K_1 M_1 + \cdots + K_l M_l$$

The probability (over random keys) that two distinct messages map to the same value for this hash function is $1/|\mathbb{F}|$.

Although in [11] this construction has been credited to Carter and Wegman [10], Bernstein in [3] mentions that this construction appears in an earlier work by Gilbert, MacWilliams and Sloane [12], in the language of finite geometries. However, this basic idea of MLHash has further been extended in [11], where they describe an efficient software implementation of this construction in a finite field $\mathbb{F}_q$ with $q = 2^{32} + 15$. Also, there are modifications to the construction to align with 32-bit word boundaries and to reduce the total number of modulo q operations.

Reducing the total number of multiplications necessary for a universal hash function has been a goal for many works in the literature. In this regard two popular schemes are pseudo dot-product (PDP) hash [26] and Bernstein-Rabin-Winograd (BRW) hash [3, 25], both of which require $n/2$ multiplications for n message blocks. These are single block hashes whereas Toeplitz construction is a d-block hash requiring $d \times n/2$ multiplications. The pseudo dot-product based hash PDP (e.g. NMH hash [11], NMH* [11], NH [13] and others [27, 28]) is defined as (for even l)

$$\mathsf{PDP}_{k_1,\cdots,k_l}(m_1, \cdots, m_l) = (m_1 + k_1)(m_2 + k_2) + \cdots + (m_{l1} + k_{l1})(m_l + k_l)$$

Another d-block universal hash, called EHC is introduced in [24] which requires $(d-1)+n/2$ multiplications for $d \leq 4$. In the same work this quantity is claimed to be the lower bound on the number of multiplications necessary to compute a universal hash over n message blocks. The main ingredient of this construction is credited to [29] in [24]. The author claims that in terms of key size and parallelizability, both Toeplitz and EHC are similar.

Another popular universal hash function is polynomial based hashing, called poly-hash [31–33] defined as follows:

The message $M = (M_1, \cdots, M_l)$ is as defined before and key K is an element over the finite field $\mathbb{F}$. The polynomial based hashing is the map:

$$\mathsf{PolyHash} : (M_1, \cdots, M_l) \xrightarrow{K} M_1.K + M_2.K^2 + \cdots + M_l.K^l$$

The probability (over random keys) that two distinct messages map to the same value for this hash function is atmost $l/|\mathbb{F}|$. One popular example is Ghash used in GCM [30].

The repeated attempts to obtain universal hash functions with high-speed software implementations led to proposals such as UMAC [13], Poly1305 [14], PolyR [15], bucket hashing [16, 17] and [18]. Design of hash functions involving linear feedback shift registers (LFSRs) has been pursued in [19–21]. Stinson [22] has also described various methods of combining hash function constructions.

A new multi-linear universal hash family has been introduced in [23]. The focus of this constructions is small hardware and other resource constrained applications. The focus of this dissertation was designing the new universal hash function. After completing its implementation, we have just started looking at its applications in different encryption and authentication schemes. In this respect, as already mentioned, our guideline is the main

paper [1] and hence, summarizing the literature on authentication and encryption has been omitted here.

Importance of the Present Work: Though the primary goal of this dissertation was initially set to implementation of the schemes described in [1], eventually, as mentioned before, the design and implementation of a suitable and competitive (in terms of speed) hash function forms the most significant contribution of this work so far. The hash functions based on BRW polynomial and Horner's rule are separately well discussed. Whereas BRW polynomial based hash function has significant advantage over other hash functions in terms of speed, its inherent recursive structure is not suitable, atleast without any nontrivial trick, to efficiently hash a variable length message with this scheme. Hence, to utilize the speed benefit given by this scheme we mixed it with Horner's rule based hash function in two levels. Now, this hash function was integrated with AES in counter mode to implement some of the schemes mentioned in the main paper. Also, the hash function was separately integrated with Salsa20 for the same purpose. For some of the cases, speed achieved from these was measured and studied comparatively. In this work two underlying fields were considered, namely the field GF(2^{128}) and the field GF(2^{256}).

The organization of the report is as follows: In Section 2 we provide an introduction to some of the basic features of our work. In Section 3 we provide our implementation of the basic field arithmetic and its different variations. In Section 4, we study the design and implementation of the hash function, we used. In Section 5, we provide implementation results of some of the schemes using AES in counter mode along with the hash function. In Section 6 the preliminary implementation of some scheme using Salsa20 along with the hash function is presented. In Section 7, we see a brief summary. We conclude the report in section 8.

2 Preliminaries

2.1 Timing Measurement:

As indicated before, speeds achieved by all the implementations in the present work have been measured and studied comparatively against the reported achievable speeds so far. For timing measurement we basically used the following piece of code along with appropriate iteration numbers for cache warming and stabilizing the result.

```
#define STAMP ({unsigned res; __asm__ __volatile__ ("rdtsc" : "=a"(res) : : "edx"); res;})

#define DO(x) do { \
int i,j; \
for (i = 0; i < M; i++) { \
unsigned c2, c1;\
for(j = 0;j < CACHE_WARM_ITER;j++) {x;}\
c1 = STAMP;\
for (j = 1; j <= N; j++) { x; }\
c1 = STAMP - c1;\
median_next(c1);\
} } while (0)
```

This framework was used to produce the results in [34,35] to report the speed of OCB3. Here, for measuring the number of CPU cycles elapsed to execute a piece of code represented by x the x86 time-stamp counter (TSC) is used. The Time Stamp Counter (TSC) is a 64-bit register present on all x86 processors since the Pentium. It counts the number of cycles since reset. The instruction *rdtsc* returns the value of TSC in $EDX : EAX$, which is read by the macro STAMP. This strategy has a drawback. The TSC read instruction might be executed out of order, in some cases it has high latency, and it continues counting when other processes run. Hence, some precautions are taken. To reduce the effect of memory access time, the set of instructions under test is executed sufficient number of times without measuring time. This is the standard way to perform cache-warming, as a result of which the instruction set is expected to reside in the cache before the timing measurement starts and as a result memory access time should not be included in the measured time. Next, the TSC is read by STAMP macro, followed by N iterations of the set of instructions (x) under test. At the end, again the TSC is read by STAMP macro and the difference of these two readings before and after the executions of x for N times gives the number of CPU cycles required for N executions of x. This entire thing is done for a number of times, M, say. For doing away with the imperfections in this measurement strategy, the median of these M readings is taken as the result and dividing this by N gives the time taken by single execution of x.

For timing measurement for this work, we have used the following environment:

- Hardware:
  - Model name : Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
  - Graphics : Intel (R) Haswell Desktop
  - L1 Cache size : 256 KB
- Software :
  - OS : Ubuntu 14.04 LTS
  - Compiler flags used : -mpclmul -mavx2 -O3 (optionally) -maes (optionally)

2.2 Intel Intrinsics: background

In compiler theory, an intrinsic function is a function available for use in a given programming language whose implementation is handled specially by the compiler. In this work we are concerned with the language C and hence the subsequent discussion will be with reference to C. Intrinsics are assembly-coded functions that allow you to use C function calls and variables in place of assembly instructions. These are expanded inline eliminating function call overhead. Providing the same benefit as using inline assembly, intrinsics improve code readability, assist instruction scheduling, and help reduce debugging. Intrinsics provide access to instructions that cannot be generated using the standard constructs of the C language. So, intrinsics must be supported by the specific compiler.
Intel(R) C compilers support a rich set of intrinsics, though not all Intel processors support all intrinsics. These C compilers enable easy implementation of assembly instructions through the use of intrinsics, popular in the name of Intel Intrinsics. Intrinsics are provided for the following instructions:

– Intel(R) Advanced Vector Extensions (Intel(R) AVX) instructions
– Intel(R) Streaming SIMD Extensions 4 (Intel(R) SSE4) instructions
– Intel(R) Supplemental Streaming SIMD Extensions 3 (SSSE3) instructions
– Intel(R) Streaming SIMD Extensions 3 (Intel(R) SSE3) instructions
– Intel(R) Streaming SIMD Extensions 2 (Intel(R) SSE2) instructions
– Intel(R) Streaming SIMD Extensions (Intel(R) SSE) instructions
– MMX Technology instructions
– Carry-less Multiplication instruction and Advanced Encryption Standard Extensions instructions
– Half-float conversion instructions

In this work we have basically used instructions available in SSE and its different extensions, though the AVX compilation flag has been used. Some terminologies and data-types associated with Intel Intrinsics are:

– __m128: With the introduction of SSE in 1999, eight 128-bits registers were added to the CPU, xmm0 through xmm7. To directly manipulate these 128 bit registers, new data-types corresponding to any primitive data type used in C were added to the Intel Intrinsics literature. The names of these new data-types start with __m128: __m128 is used for float, __m128d for double and __m128i for int, short, char.
– Throughput: Throughput is the number of processor clocks it takes for an instruction to execute or perform its calculations. An instruction with a throughput of t clocks would tie up its execution unit for that many cycles which prevents an instruction needing that execution unit from being executed. Only after the instruction is done with the execution unit can the next instruction enter.
– Latency: Latency is the number of processor clocks it takes for an instruction to have its data available for use by another instruction. Therefore, an instruction which has a latency of l clocks will have its data available for another instruction that many clocks after it starts its execution.
  Note: Latency and throughput are typically used as the basis for instruction performance on a microprocessor.

Here are some instructions which we have used most frequently and their brief introduction:

– __m128i _mm_clmulepi64_si128 (__m128i a, __m128i b, const int $imm8$): This is the intrinsic for the instruction *pclmulqdq*.
  It performs a carry-less multiplication of two 64-bit integers, selected from a and b according to $imm8$.
  In the Haswell architecture it has latency 7 and throughput 2.
  The compilation flag required to compile codes containing this instruction is -mpclmulqdq.
  The pseudo-code for this instruction is as follows:

```
IF (imm8[0] = 0)
        TEMP1 := a[63:0];
ELSE
        TEMP1 := a[127:64];
```

```
FI
IF (imm8[4] = 0)
TEMP2 := b[63:0];
ELSE
TEMP2 := b[127:64];
FI

FOR i := 0 to 63
 TEMP[i] := (TEMP1[0] and TEMP2[i]);
 FOR j := 1 to i
TEMP [i] := TEMP [i] XOR (TEMP1[j] AND TEMP2[i-j])
 ENDFOR
 dst[i] := TEMP[i];
ENDFOR
FOR i := 64 to 127
 TEMP [i] := 0;
 FOR j := (i - 63) to 63
 TEMP [i] := TEMP [i] XOR (TEMP1[j] AND TEMP2[i-j])
 ENDFOR
 dst[i] := TEMP[i];
ENDFOR
dst[127] := 0
```

- __m128i _mm_xor_si128 (__m128i a, __m128i b): This is the intrinsic for the instruction *pxor*.
  It computes the bitwise XOR of 128 bits (representing integer data) in a and b.
  In the Haswell architecture it has latency 1 and throughput 0.33.

- __m128i _mm_set_epi8 (char $e15$, char $e14$, char $e13$, char $e12$, char $e11$, char $e10$, char $e9$, char $e8$, char $e7$, char $e6$, char $e5$, char $e4$, char $e3$, char $e2$, char $e1$, char $e0$): It sets packed 8-bit integers in the target with the supplied values in reverse order.
- __m128i _mm_set_epi32 (int $e3$, int $e2$, int $e1$, int $e0$): It sets packed 32-bit integers in the target with the supplied values.
- __m128i _mm_slli_si128 (__m128i a, int $imm8$): This is the intrinsic for the instruction *pslldq*.
  It shifts a left by $imm8$ bytes while shifting in zeros, and store the results in the target.
  In Haswell architecture it has latency 1 and throughput 0.5.
- __m128i _mm_srli_si128 (__m128i a, int $imm8$): This is the intrinsic for the instruction *psrldq*.
  It shifts a right by $imm8$ bytes while shifting in zeros, and store the results in the destination.
  In Haswell architecture it has latency 1 and throughput 0.5.
- __m128i _mm_shuffle_epi32 (__m128i a, int $imm8$): This is the intrinsic for the instruction *pshufd*.
  It shuffles 32-bit integers in a using the control in $imm8$, and store the results in the

destination.
In Haswell architecture it has latency 1 and throughput 1.
The psedo-code for this instruction is as follows:

```
SELECT4(src, control){
   CASE(control[1:0])
   0: tmp[31:0] := src[31:0]
   1: tmp[31:0] := src[63:32]
   2: tmp[31:0] := src[95:64]
   3: tmp[31:0] := src[127:96]
   ESAC
   RETURN tmp[31:0]
}
dst[31:0] := SELECT4(a[127:0], imm8[1:0])
dst[63:32] := SELECT4(a[127:0], imm8[3:2])
dst[95:64] := SELECT4(a[127:0], imm8[5:4])
dst[127:96] := SELECT4(a[127:0], imm8[7:6])
```

3 Implementation of Basic Field Arithmetic: It's variations

Let us start the discussion with the case of $GF(2^{128})$. We started with a routine to implement the field arithmetic for GF(2^{128}). While addition is simply bitwise xor and is not a costly operation, for reducing the cost of multiplication we tried out four methods and measured their speed comparatively. Any finite field is generated modulo an irreducible polynomial. If the field is $GF(2^n)$, the polynomial must be a binary polynomial of degree n. For implementational cost issues its always better to have a polynomial, satisfying all the required properties, with as less number of non-zero coefficients as possible. We take the irreducible polynomial $\mathbf{x^{128} + x^7 + x^2 + x + 1}$ to be defining the field $GF(2^{128})$, as it satisfies all the above properties and is used in [36]. As a result, we can use the algorithms of [36] as it is. Now, multiplication of two field elements consists of two steps: the first one is usual multiplication of the elements (if the field is $GF(2^n)$, then it should be carryless multiplication as in this types of fields 1+1 = 0). The second step is to reduce this product modulo the defining irreducible polynomial of the field.

3.1 Field multiplication:

Now, for only multiplication of two 128-bits string, we can use either the normal Schoolbook algorithm or the asymptotically faster Karatsuba algorithm. In modern Intel processors, especially in Haswell architecture enabled ones, the intrinsic instruction PCLMULQDQ for carryless multiplication of two 64-bit quantity gives significant speed advantage over other available mechanisms for the same so far. Hence, we have used this in our work.
Let a and b represent two 128-bit quantities to be multiplied. As we used PCLMULQDQ which multiplies two 64-bit quantity to give their carryless product, let the bit-string representation of a and b be divided in lower 64 bit and upper 64 bit parts. Let the parts be respectively a_u and a_l for a and b_u and b_l for b. So, a can be written as $a = (a_u \times 2^{64} + a_l)$;

similarly b can be written as $b = (b_u \times 2^{64} + b_l)$. Now, the Schoolbook method multiplies a and b like this:

$$a \times b = (a_u \times 2^{64} + a_l) \times (b_u \times 2^{64} + b_l) = (a_u \times b_u) \times 2^{128} + (a_u \times b_l) \times 2^{64} + (b_u \times a_l) \times 2^{64} + (a_l \times b_l)$$

So, along with some additions and shifts, it requires *four* applications of PCLMULQDQ, all of which are independent of each other and hence scheduling them consecutively reduces the effective overhead of latency period. As the latency (7 cpu cycles) dominates over throughput (2 cpu cycles) of PCLMULQDQ, this type of scheduling improves the overall speed via instruction pipelining.
Now, the Karatsuba method does this by:

$$a \times b = (a_u \times 2^{64} + a_l) \times (b_u \times 2^{64} + b_l)$$
$$= (a_u \times b_u) \times 2^{128} + (a_u + a_l) \times (b_u + b_l) - (a_u \times b_u) - (a_l \times b_l) \times 2^{64} + (a_l \times b_l)$$

So, although this method needs some extra additions compared to that in Schoolbook method, it requires only *three* multiplications, all of which are again independent of each other and hence their consecutive scheduling improves the overall speed.
Here, we need to reduce the 255-bit (as carryless) product obtained by one of the previous methods modulo the polynomial $x^{128} + x^7 + x^2 + x + 1$. For this reduction we experimented two strategies, one of them being the reduction by multiplication strategy as described in [4] and another one is a modified version of the strategy to reduce by shifts (applicable only in case of this particular polynomial) mentioned in the same paper.
The procedure of reducing the obtained product of degree 254 modulo $g = x^{128} + x^7 + x^2 + x + 1$ via multiplication is as follows:

Reduction by multiplication: Consider the product polynomial as a 256-bit string with most significant bit (msb) set to zero. Let us split it to two 128-bit halves. The least significant half just need to be xored with the final remainder as the degree of g is 128. So, if the polynomial corresponding to the most significant half is denoted by $c(x)$, we need an efficient procedure to calculate

$$p(x) = c(x) \cdot x^t \ mod \ \ g(x) \tag{1}$$

Where,

- degree of $c(x)$ is $s - 1$. Here $s = 128$.
- degree of g is t. Here $t = 128$.
- so, $t = s$.

Let us use the notation $L_u(v)$ to denote the coefficients of the u least significant terms of the polynomial v and use $M^u(v)$ to denote the coefficients of its u most significant terms. The polynomial $p(x)$ can be expressed as:
$p(x) = c(x) \cdot x^t \ mod \ g(x) = g(x) \cdot q(x) \ mod \ x^t$

where $q(x)$ is a polynomial of degree $s-1$ which is the quotient from the division of $c(x) \cdot x^t$ by g. The dividend $c(x) \cdot x^t$ can be expressed as

$$c(x) \cdot x^t = g(x) \cdot q(x) + p(x) \tag{2}$$

From (2), its evident that the t least significant terms of the polynomial $g \cdot q$ are equal to terms in p. Hence,

$$p(x) = g(x) \cdot q(x) \ \ mod \ \ x^t = L^t(g(x) \cdot q(x)) \tag{3}$$

Now, let us define by g^* the t least significant terms of the polynomial g. So, 2 can be written as:

$$p(x) = L^t(q(x) \cdot g^*(x)) \tag{4}$$

So, we need to know q in order to compute $p(x)$.

$$2 \Leftrightarrow c(x) \cdot x^{t+s} = g(x) \cdot q(x) \cdot x^s + p(x) \cdot x^s \tag{5}$$

Let

$$x^{t+s} = g(x) \cdot q^+(x) + p^+(x) \tag{6}$$

where, q^+ is an s-degree polynomial and p^+ is a $(t-1)$ degree polynomial.So,

$$5 \ \ and \ \ 6 \Leftrightarrow M^s(c(x) \cdot g(x) \cdot q^+(x) + c(x) \cdot p^+(x)) = M^s(g(x) \cdot q(x) \cdot x^s + p(x) \cdot x^s) \tag{7}$$

Now, it is to be noted that the polynomials $c \cdot g \cdot q^+$ and $g \cdot q \cdot x^s$ are of degree $t+2s-1$, the polynomial $c \cdot p^+$ is of degree $t+s-2$ and the polynomial $p \cdot x^s$ is of degree $t+s-1$. Hence,

$$M^s(c(x) \cdot g(x) \cdot q^+(x)) = M^s(g(x) \cdot q(x) \cdot x^s) \Rightarrow M^s(g(x) \cdot M^s(c(x) \cdot q^+(x)) \cdot x^s) = M^s(g(x) \cdot q(x) \cdot x^s) \tag{8}$$

which is satisfied for q given by

$$q = M^s(c(x) \cdot q^+(x)) \tag{9}$$

So, p can be found by

$$p(x) = L^t(g^*(x) \cdot M^s(c(x) \cdot q^+(x))) \tag{10}$$

where $g^*(x)$ and $c(x)$ are easily available. The interesting thing to be noted is that when x^{256} is divided by the particular g we consider here the quotient is g itself, hence giving q^+ also easily. Thus (10) implies that the main operation in doing the reduction modulo this particular g is two 128-bit multiplications. This means we need to use PCLMULQDQ atleast six times for this reduction. But investigating (10) closely, we can reduce it to three, one for $(c(x) \cdot q^+(x))$ part which must be done first and the other two owing to the other multiplication depend on the first one. The next two are independent of each other, hence can be scheduled consecutively in any order.

Reduction by shift: The reduction can be sped up if the particular form of g is taken into account. This g can be represented as the bit sequence $[1 : \langle 120zeros \rangle : 10000111]$. Multiplying this carry-less with a 128-bit value and keeping the 128 most significant bit can be obtained by:
(i) Shifting the 64 most significant bits of the input by 63, 62 and 57-bit positions to the right.
(ii) xoring these shifted copies with the 64 least significant bits of the input.
Next, we carry-less multiply this 128-bit result with g, and keep the 128 least significant bits. This can be done by:
(iii) shifting the 128-bit input by 1, 2 and 7 positions to the left.
(iv) xoring the results.
We have achieved the same effects with slightly modified actions in our work for reducing total number of bit-shifts in our work. This algorithm requires no multiplication, but only shifts and xor.

3.2 Batch multiplication:

Batch multiplication is one of the things which we have studied in considerable detail in this work. When there are a number of independent 128-bit multiplications, we can exploit this independence for speed improvement. Each of the 128-bit multiplications consists of three (for Karatsuba method) or four (for Schoolbook method) independent 64-bit multiplications. So, if the number of independent 128-bit multiplications at some point of the implementation is n, effectively we have $3 \times n$ or $4 \times n$ independent 64-bit multiplications in our hand. Instead of considering these multiplications separately, we can schedule them in such a way so that the overall latency for a group of these multiplications is reduced. This strategy is called multiplication in batches. The reason behind reduced latency is further explained below.

PCLMULQDQ is used for each of these 64-bit multiplications. As mentioned earlier, in Haswell architecture PCLMULQDQ has latency as 7 cpu cycles and throughput as 2 cpu cycles. So, from the moment when a multiplication operation is initiated, after 7 cpu cycles the first bit of output is generated. If the second PCLMULQDQ instruction is triggered only after the entire output for first PCLMULQDQ has been generated, then another span of 7 cpu cycles is needed to get the next bit of output. This overall waiting time could be reduced, if the second PCLMULQDQ instruction had been triggered just in the next cpu cycle after the first one has been triggered. In this case, in fact we would have to wait for only 7 cycles for the first bit of output. This strategy can be used in more efficient way by batching more number of multiplications together by reducing the overall waiting time for all of them to that for just one. But, all of these batched multiplications must be independent of each other to get the desired speed up, otherwise the dependent one will anyway have to wait for the result of the PCLMULQDQ instructions on which it depends. This strategy can be exploited more efficiently here as we must break up each 128-bit multiplications into several 64-bit multiplications. The reason is the availability of efficient instruction PCLMULQDQ for performing 64-bit product and absence of any such instruction for 128-bit multiplication. As two independent 128-bit multiplications implies

invoking PCLMULQDQ at-least six times independently it implies more batching. Here we tried upto four independent 128-bit multiplications in a batch. Why not more?

Doing more multiplications in batch should lead to speed up, but, the effect may become less noticeable if the batch size increases too much. If the batch size is such that the latency time of the first PCLMULQDQ is saturated, i.e. if already seven independent PCLMULQDQ instructions have been scheduled in the consecutive seven CPU cycles, then from the eighth instruction onward will wait for the completion of first batch execution. Again, at most next seven PCLMULQDQ instructions can fill up the latency time of the eighth one and from fifteenth onward will have to wait for third slot. This may result in less noticeable speed up. Hence we tried here upto a batch of four independent 128-bit multiplications and experimenting with larger batch sizes can be taken up as a topic of further study.

Now, all these combinations for multiplication followed by reduction are comparatively studied for speed and the results are as follows:

Strategy	Size of Batch			
	1	2	3	4
Schoolbook-mult	72.036	55.205	48.748	44.401
Schoolbook-shift	142.325	131.403	132.254	124.362
Karatsuba-mult	87.342	75.311	62.907	55.48
Karatsuba-shift	162.629	142.182	146.92	140.259

Table 1. Number of cycles per 128-bit multiplication for different options for basic and batched multiplications

Now, let us discuss the $GF(2^{256})$ case, especially stressing upon the differences of it with the previous case:

For 256-bit case: We worked in parallel for $GF(2^{256})$. So that we don't need to re-implement all the things again for 256-bit case, we implemented an outer layer so that choosing between 128-bit case and 256-bit case could be done with minimum effort. This is done with the help of a header file, which chooses appropriate multiplication routine, some basic operations and basic data-types depending on the value set to a variable, indicating the correct field. Hence, we implemented only the multiplication routine separately for 256-bit case.

The first basic issue to note for $GF(2^{256})$ is that there is no support in Intel intrinsic for 128-bit multiplication as is there for 64-bit multiplications (PCLMULQDQ). Hence, as we divided each 128-bit multiplications into 64-bit ones, here we cannot break 256-bit multiplications into 128-bit ones and stop. We have to break each 256-bit multiplications into corresponding 64-bit multiplications.

For carryless multiplication of two 256-bit numbers by Karatsuba method, we need *three* independent 128-bit multiplications, each of which can be further broken into three independent 64-bit multiplications by Karatsuba method. Hence, we get *nine* independent 64-bit multiplications, each of which is done by PCLMULQDQ. So, if we have n independent 256-bit multiplications at some point in the implementation, we effectively get $9 \times n$ independent use of PCLMULQDQ instructions, which can be scheduled in batches again for batch multiplication.
Similarly, by Schoolbook method n independent 256-bit multiplications at any point in implementation means $16 \times n$ independent 64-bit multiplications. Hence, these $16 \times n$ invoking of PCLMULQDQ can be scheduled in batches, so that speed improvement is achieved due to batch multiplication.

In this case, we consider the irreducible polynomial $x^{256} + x^{10} + x^5 + x^2 + 1$ of degree 256. For reducing by this polynomial we used the previous two algorithms, extended to the 256-bit case. Here, the reduction by multiplication needs four invoking of PCLMULQDQ, out of which two depend on the other two. Let us see this method in detail:

The reduction by multiplication algorithm for 128-bit case applies similarly to this case with some obvious modifications, like $s = 256, t = 256$ and g is the irreducible polynomial of degree 256, as mentioned before. Now, we need to follow equation (10). Here, $c(x)$ is 255-degree polynomial and $q^+(x)$ is the polynomial corresponding to g (for the same reason as in the case of 128-bit). We need the most significant 256-bits of their product. Let $c(x)$ be denoted by $y_1||y_0$, where y_1 is the most significant 128-bit part and y_0 is the least significant 128-bit part. Similarly $q^+(x)$ can be denoted by $1||g^*$ where, g^* is the least significant 255 bits of g. Let, g_0 denote the least significant 128 bits of g. Hence, equation (10) can be calculated by the following method:

$$
\begin{aligned}
c &= y_1||y_0 \\
q^+ &= 1||\langle 128\ 0\text{'s}\rangle||g_0 \\
&= (1 \times 2^{256} + \langle \text{least significant 256 bits of g}\rangle) \\
M^{256}(c.q^+) &= M^{256}(c \times (1 \times 2^{256} + \langle \text{least significant 256 bits of g}\rangle)) \\
&= M^{256}(c \times 2^{256} + c \times \langle \text{least significant 256 bits of } g\rangle) \\
&= c + M^{256}(c \times \langle \text{least significant 256 bits of } g\rangle) \\
&= c + M^{256}((y_1||y_0) \times (\langle 128\ 0\text{'s}\rangle||g_0)) \\
&= c + M^{256}((y_1 \times g_0) \times 2^{128} + (y_0 \times g_0)) \\
&= (y_1||y_0) + (\langle 128\ 0\text{'s}\rangle||\langle \text{most significant 128 bit of } (y_1 \times g_0)\rangle) \\
&= y_1||(y_0 + \text{most significant 128 bit of} \\
&\quad (y_1 \times (\langle 64\ 0\text{'s}\rangle||\langle 64 \text{ least significant bit of } g_0\rangle))) \\
&= y_1||(y_0 + \text{most significant 128 bit of} \\
&\quad (\langle 64 \text{ most significant bit of } y_1\rangle \times \langle 64 \text{ least significant bit of } g_0\rangle \times 2^{64} \\
&\quad + \langle 64 \text{ least significant bit of } y_1\rangle \times \langle 64 \text{ least significant bit of } g_0\rangle)) \\
&= y_1||(y_0 + (\langle 64\ 0\text{'s}\rangle||\langle 64 \text{ bit right shift of} \\
&\quad (\langle 64 \text{ most significant bit of } y_1\rangle \times \langle 64 \text{ least significant bit of } g_0\rangle)\rangle)) \qquad (1) \\
&= y_1||temp_1(say) \\
L^{256}(g^* \times M^{256}(c.q^+)) &= L^{256}(g^* \times (y_1||temp_1)) \\
&= L^{256}((\langle 128\ 0\text{'s}\rangle||g_0) \times (y_1||temp_1)) \\
&= L^{256}((g_0 \times y_1) \times 2^{128} + (g_0 \times temp_1)) \\
&= \langle \text{least significant 128-bits of } (g_0 \times y_1)\rangle + (g_0 \times temp_1) \\
&= \langle 128 \text{ bit left shift of } (g_0 \times y_1)\rangle + (g_0 \times temp_1) \qquad (2)
\end{aligned}
$$

Here, for (1) we need one use of PCLMULQDQ and for (2) there are three uses of PCLMULQDQ: one for another part of $g_0 \times y_1$, which is not needed in (1) and two for two nontrivial 64-bit multiplications in $g_0 \times temp_1$ (note that most significant 64-bit of g_0 is all-zero, giving rise to two trivial 64-bit multiplications). Out of these four 64-bit multiplications, first two are independent of others and hence can be scheduled at the very beginning of the corresponding code and can be scheduled consecutively for better performance. The last two depend on $temp_1$, i.e. the result of the first one, but they are independent of each other. Hence, once $temp_1$ is calculated these last two multiplications can be scheduled in any order and specifically can be scheduled consecutively for better performance.

The reduction by shift algorithm is naturally extended to 256-bit case, which does not require any multiplication similar to the 128-bit case.

As indicated earlier, in 256-bit case also we tried for batch multiplication upto batch-size of four. Here, scope for batching is more, as there are more number of independent

multiplications than in the case of 128-bit. But as previously discussed, the speed up is naturally less noticeable.

The timing results are as follows:

Strategy	Size of Batch			
	1	2	3	4
Schoolbook-mult	170.398	140.825	137.366	140.028
Schoolbook-shift	334.82	326.386	320.869	316.299
Karatsuba-mult	205.542	181.107	181.022	179.043
Karatsuba-shift	358.8909	360.537	361.068	355.775

Table 2. Number of cycles per 256-bit multiplication for different options for basic and batched multiplications

4 Design and Implementation of the two-level Hash Function

Let $\{\text{Hash}_\tau\}$ be a family of keyed hash function on a common domain and a common range, where the key τ is chosen from an appropriate finite set. Two kinds of probabilities are defined for the hash function family:
Collision Probability: For all distinct x and x$'$, the collision probability of Hash_τ corresponding to the pair (x, x$'$) is $\Pr[\text{Hash}_\tau(x) = \text{Hash}_\tau(x')]$, where the probability is taken over the uniform random choice of τ.
Differential Probability: For all distinct x and x$'$ and any y, the differential probability of Hash_τ corresponding to the triplet (x, x$'$, y) is $\Pr[\text{Hash}_\tau(x) \oplus \text{Hash}_\tau(x') = y\ ]$, where the probability is taken over the uniform random choice of τ.
If all the collision probabilities are bounded above by ϵ, then Hash_τ is said to be almost universal (ϵ-AU); if all the differential probabilities are bounded above by ϵ, then Hash_τ is said to be almost XOR universal(ϵ-AXU). It is to be noted here that, this ϵ in any of the two cases need not be an absolute constant. For many hash functions this is a constant, which depends on the length of the input message.
The two issues regarding hash functions that
1. A hash function need not be defined for all possible input lengths and
2. Collision and differential probabilities of a hash function can be guaranteed to be low only for equal length inputs
can be tackled by suitable padding technique, as a result of which we get a single input hash function without any restriction on input length and with low collision and differential probabilities for any two inputs. There are well known examples of both AU and AXU (for equal length strings) single-input hash functions, from which vector input hash functions which are AXU for variable length vectors can be constructed and some concrete constructions for doing so have been described in the main paper.
As the basis of this construction, an efficient single-input hash function having the AU and AXU properties(for equal length strings) are needed. A family of fast (single-input) hash functions have been proposed by Bernstein based on an earlier work by Rabin and Wino-

grad and the family has been called the BRW functions in [4]. This hash function satisfies these properties. Moreover, Horner's rule based hash function is well-known and satisfies these properties. Both of them are used in the design of our Hash function.

4.1 Overview:

In this and subsequent sections, we use the following notation. Given a binary string S, let len(S) denote the length of S, i.e., len(S) is the number of bits in S. Given an integer i with $0 \leq i \leq 2^n - 1$, let $\mathsf{bin}_n(i)$ denote the n-bit binary representation of i.

Here, we have designed a new universal hash function. Let, the domain or input message space of the hash function be denoted by $\mathcal{M}$ and the range or digest space be denoted by $\mathcal{R}$. Let, the key space be denoted by $\mathcal{K}$. Here,
$\mathcal{M} : \{0,1\}^i$ (i is any positive integer)
$\mathcal{R} : \{0,1\}^n$
$\mathcal{K} : \{0,1\}^{2n}$
In this work we consider two values of n, i.e. 128 and 256 separately. Initially, we take two keys, each of length n-bit, independent of each other. These are the keys for the two levels of our hash function. Later in some of schemes some tricks have been used to reduce the requirement of the number of independent bits.

If needed the input message is padded by 0's to make its length divisible by n, i.e. if already it is not so. Then, it is divided into stream of strings each of length n, each of which represents an element of underlying the field $GF(2^n)$. Each such string is called a *block* here. This work uses a concept of *superblock* also, which represents a number of blocks. We have considered that a superblock consisting of 31 blocks is a complete superblock and a superblock consisting of lesser number of blocks is a partial superblock.

Our hash function is a two-level one, the first level of which is the BRW hash function. As BRW functions can be implemented very efficiently when the length of the input message is known beforehand, we have used 31-block BRW in the initial level. After padding the input message if needed, the message is divided into superblocks, the last piece being possibly a partial superblock. Now, we compute a digest for each of the superblocks after applying the BRW based hash on it. For all other superblocks except the last one we apply the BRW hash for 31 blocks. For the last one, if it's a complete superblock we apply the same. If it's not and contains $1 \leq i \leq 30$ blocks, we apply the BRW hash corresponding to i blocks on it to compute the digest. As a result we get a stream of digests, each of which is again a n-bit quantity.

Now, for security guarantee we pad this stream of digests with the initial length (before padding) of the message expressed as a n-bit quantity, i.e. if the input message is M, here padding is done by $\mathsf{bin}_n(len(M))$. These n-bit blocks are now fed into Horner's rule based hash function to get the final n-bit digest. For these two levels of hashing two independent keys has initially been chosen randomly. Later in some cases some tricks have been used.

4.2 Lower level in 2-step hash: BRW based hashing

In [3], Bernstein has defined a family of polynomials based on a previous work by Rabin and Winograd. This family of polynomials has been named as BRW polynomials in [4]. The definition of the family is as follows:
Definition of BRW polynomials: The polynomial $H(M_1, M_2, \cdots, M_l) \in \mathbb{F}[x]$, ($\mathbb{F}$ is the underlying finite field) is defined as follows for $l \geq 0$:

- H() = 0
- $H(M_1) = M_1$
- $H(M_1, M_2) = M_1 x + M_2$
- $H(M_1, M_2, M_3) = (x + M_1)(x^2 + M_2) + M_3$
- $H(M_1, M_2, \cdots, M_l) = H(M_1, \cdots, M_{t-1})(x^t + M_t) + H(M_{t+1}, \cdots, M_l)$ if $t \in 4, 8, 16, 32, \cdots$ and $t \leq l < 2t$

When this family of polynomials is used for hashing, then x denotes the corresponding key of hashing. This key is chosen randomly from the underlying field. The polynomial evaluated at the particular x is denoted as H_x.

These polynomials on l message blocks defined over $\mathbb{F}$ have the interesting property that they can be used for authentication, but, for $l \geq 2, H_x(M_1, \cdots, M_l)$ can be computed using $\lfloor l/2 \rfloor$ multiplications and $\lceil \lg l \rceil$ squarings.

As discussed in [9], a BRW polynomial $H_x(M_1, \cdots, M_l)$ can be represented as a tree T_l which contains three types of nodes, namely, multiplication nodes, addition nodes and leaf nodes. This BRW tree can be recursively constructed using the following rules:
1) For $l = 2, 3$ it is trivial.
2) If $l = 2^s$, for some $s \geq 2$, the root of T_l is a multiplication node. The left subtree of the root consists of a single addition node which in turn has the leaf nodes h^l and X_l as its right and left child, respectively. The right subtree of the root is the tree T_{l-1} .
3) If $2^s < l < 2^{s+1}$ for some $s \geq 2$, the root is an addition node with its left subtree as T_{2^s} and the right subtree as T_{l-2^s} .

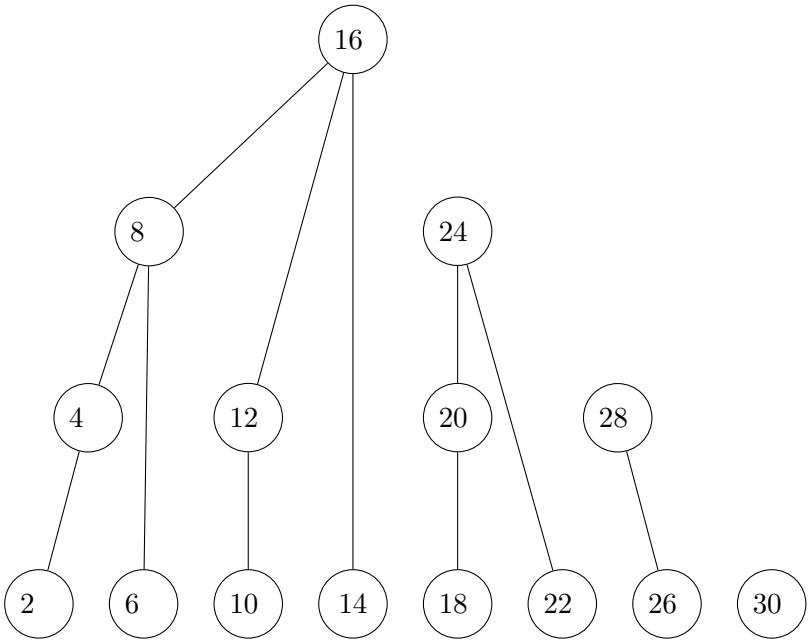
According to this construction, the following important property of BRW trees is observed:
For a multiplication node, either, its left child is labeled by a message block X_j and the right child is labeled by h; or, its left child is an addition node which in turn has a message block X_j and h^k as its children for some j and k. As a consequence, for a multiplication node, there is exactly one leaf node in its left subtree which is labeled by a message block.

As indicated in [9], as we are only interested in multiplications, we can ignore the addition nodes and thus simplify the BRW tree by deleting the addition nodes from it and reduce the tree T_l corresponding to the polynomial $H_x(M_1, \cdots, M_l)$ to a new tree by applying the following steps in sequence.
1) Label each multiplication node v by j where X_j is the leaf node of the left subtree rooted at v.
2) Remove all nodes and edges in the tree T_l other than the multiplication nodes.

3) If u and v are two multiplication nodes, then add an edge between u and v if u is the most recent ancestor of v in T_l .
The resulting structure is a forest, which is called a collapsed forest, denoted by F_l. It can be shown that, there is a unique multiplication associated with each node of a collapsed forest. The structure of the collapsed forest corresponding to a polynomial $H_x(\cdot)$ helps us to visualize the dependencies of the various multiplications involved in the computation of $H_x(\cdot)$.

If there is no path between two nodes in the forest, it implies that the corresponding multiplications are independent of each other. If level of a node v in the collapsed forest is the number of nodes present in the longest path from v to a leaf node, then the nodes with level 1 are independent and any node with level more than 1 are dependent on some other multiplication. Hence, at first stage any of the independent multiplications can be performed in any order in a batch, resulting in an increased efficiency due to pipelining. Any of the level 2 multiplications can be started only when all the level 1 multiplications, on which this level 2 multiplication depend, have been completed and so on. As a result, if a pipelined multiplier with N stages are used, then as long as N independent(or dependent on multiplications, which are already completed) multiplications are available at any point of time during the computation, the efficiency is increased. The rigorous scheduling strategy, which will maximize this efficiency for computation of BRW polynomial for any number of blocks is discussed in the same paper.

We have used the 31-block BRW hash function for the lower level of our two-level hash function. The implementation of BRW hash function was as follows:
Let the length of the input message to the lower level is B bytes. So, the number of n-bit blocks in it is $\frac{B\times 8}{n}$. Now, we divide this message in superblocks containing 31-blocks, with possibly the last superblock not consisting of as much as 31-blocks. So, we get $\lceil \frac{B\times 8}{n\times 31} \rceil$ superblocks. Now, according to the definition of BRW-polynomial in [3] we implement the digest-evaluation routine for 31-blocks and also a separate digest evaluation routine for superblocks consisting of any number of blocks between 1 to 30, for evaluating the last superblock in our message, if it does not contain 31 blocks.
The BRW tree for 31-block BRW polynomial is as follows:



From the above diagram it is clear that, initially there are *eight* independent multiplications, which can be scheduled in any order in the first slot. Once the multiplications corresponding to nodes labeled as 2, 10, 18, 26 are completed any of those corresponding to nodes labeled as 4, 12, 20, 28 can be scheduled and so on. In primary (sequential) implementation of this 31-block BRW polynomial evaluation these multiplications and other associated operations are scheduled in any order compatible with the form of the polynomial, i.e. it is ensured that the operations on which the present operation is dependent are already complete. But, the batched multiplication routines have also been used here for pipelining the independent multiplications available at any stage.

Batching in BRW hash evaluation: We utilized the pipelined multiplication routines we already implemented, in the BRW digest evaluation of 31-blocks inputs in the following manner. We need total 15 multiplications in 31-block BRW evaluation, as is evident from the tree. In pipelining upto level 2, we tried to group them in bundles of 2 multiplications whenever possible and implemented these using the routine for two multiplication at a time. So along with other necessary associated operations, we performed the bundles of multiplications in the following order:
$\{2, 6\}, \{10, 14\}, \{18, 22\}, \{26, 30\}, \{4, 12\}, \{20, 28\}, \{8, 24\}$ and at last the single multiplication $\{16\}$.
Similarly, for pipelining upto level 3, we grouped the multiplications in bundles of 3 and used the routine for 3 multiplications at a time. The multiplications ordering was like this:
$\{2, 6, 10\}, \{14, 18, 22\}, \{26, 30, 4\}, \{20, 12, 8\}, \{28, 24, 16\}$.
Similarly, for level 4 pipelining the multiplications ordering was:
$\{2, 6, 10, 14\}, \{18, 22, 26, 30\}, \{4, 12, 20, 28\}, \{8, 24\}, \{16\}$.
Clearly, only level 3 pipelining is nice in structure and does not require mixing with any

other level. The timing results are as follows:

Strategy	Level of Pipelining(cycles per byte)			
	1	2	3	4
Schoolbook-mult	0.54	0.57	0.52	0.54
Schoolbook-shift	0.56	0.57	0.52	0.53
Karatsuba-mult	0.56	0.57	0.51	0.55
Karatsuba-shift	0.55	0.55	0.52	0.52

Table 3. Number of cycles required for BRW based Hash(128-bit) for one complete superblock: with different strategies of multiplication and different levels of pipelining

As the last superblock can contain i blocks, where $0 < i < 31$, we needed to implement a routine that evaluate the BRW polynomial for this superblock. Here, we kept routines to evaluate BRW polynomial starting from 1 block to 30 blocks. Here, we used batching for multiplications only when the natural form of polynomial trivially suggests that; otherwise we used sequential instructions, as the number of times this routine will be used in computing digest for the input message is at most one.

Let us define the following function:

$$\mathsf{BRW}_x(M) = H_x(M_1, M_2, \cdots, M_l), \text{ where } l = \frac{\mathsf{len}(M)}{n}$$

Essentially, M is a superblock here, either full or partial.

4.3 Upper level in 2-step hash: Horner's rule based hashing

At this stage, we have a stream of n-bit blocks as input to the second level of our hash function, which is Horner's rule based hash. We have also a key of length n-bit chosen randomly from $GF(2^n)$.

Let us recollect the Horner's rule based hash function.
Let $f(x)$ be a polynomial of degree d defined as:
$f(x) = f_d x^d + f_{d-1}x^{d-1} + \cdots + f_1 x + f_0$
For any x this can be evaluated at x by d multiplications as:

$$(((f_d x + f_{d-1})x + f_{d-2})x + \cdots + f_1)x + f_0 \tag{11}$$

This evaluation technique of the above polynomial using only d multiplications and d additions is famous in the name of Horner. For hashing a stream of n-bit blocks, a n-bit element is randomly chosen from $GF(2^n)$ and used as x in the above formula. This x is called the key in this hashing context.

The input stream is considered as the stream of n-bit coefficients f_i taken in consecutive order, in the above scenario. Now, the polynomial is evaluated at that particular key, which gives the final digest of our hash function.

Batching in Horner Rule's based hash function evaluation: Now, the Horner's rule based hash function discussed above when implemented in its simplest form gives a routine, which we call Horner1 here. Then we tried for pipelining (called decimation in this context) in this routine too and got Horner2, Horner3 and Horner4 depending on the level of decimation. The scheme with p level of decimation is as follows, for any integer $p \leq d$:
$a_0 + a_1x + a_2x^2 + \cdots + a_dx^d$
$= (a_0+a_px^p+a_{2p}x^{2p}+\cdots)+(a_1+a_{p+1}x^p+a_{2p+1}x^{2p}+\cdots)x+\cdots+(a_{p-1}+a_{2p-1}x^{2p}+\cdots)x^{p-1}$

Now for evaluating the portion inside each of () we apply formula (11). When the level of decimation is p, p such expressions inside () need to be evaluated, which means we can do p multiplications, one for each of (), together.
When we opt for 2-decimation Horner, the expression is evaluated as:

$$(a_0 + a_2x^2 + a_4x^4 + \cdots) + (a_1 + a_3x^2 + a_5x^4 + \cdots)x$$

If the degree of the overall polynomial is d, then the above expression can be written as:
$(a_0 + a_2x^2 + a_4x^4 + \cdots + a_dx^d) + (a_1 + a_3x^2 + a_5x^4 + \cdots + a_{d-1}x^{d-2})x$, if d is even
Or,
$(a_0 + a_2x^2 + a_4x^4 + \cdots + a_{d-1}x^{d-1}) + (a_1 + a_3x^2 + a_5x^4 + \cdots + a_dx^{d-1})x$, if d is odd.
$\Updownarrow$
$(((a_dx^2 + a_{d-2})x^2 + \cdots + a_2)x^2 + a_0) + (((a_{d-1}x^2 + a_{d-3})x^2 + \cdots + a_3)x^2 + a_1)x$, if d is even
Or,
$(((a_{d-1}x^2 + a_{d-3})x^2 + \cdots + a_2)x^2 + a_0) + (((a_dx^2 + a_{d-2})x^2 + \cdots + a_3)x^2 + a_1)x$, if d is odd.
So, in each level we need two independent multiplications, like $\{a_d * x^2\}$ and $\{ad - 1 * x^2\}$, $(a_dx^2+a_{d-2})*x^2$ and $(a_{d-1}x^2+a_{d-3})*x^2$ or, $(a_{d-1}x^2+a_{d-3})*x^2$ and $(a_dx^2+a_{d-2})*x^2$. Here, we can use our routine for two multiplications. At last the result of second sequence (i.e the expression inside second()) is multiplied by x and is added with the result of first sequence. Similarly, the 3-decimation and 4-decimation Horner's rule based hash is implemented. Boundary conditions in each of the decimations is handled properly. The timing results are as follows:

Let us now define the following function:

$\mathsf{Horner}_x(f_d, \cdots, f_0) = (((f_dx+f_{d-1})x+f_{d-2})x+\cdots+f_1)x+f_0,$ where f_i's, for $0 \leq i \leq d$, are n-bit blocks.

4.4 The Complete two-level hash function:

Now, that all components of the hash functions are ready to be integrated, we implement the total hash function. Let the input message M contains B' bytes and the pair of ran-

Strategy	Level of decimation(cycles per byte)			
	1	2	3	4
Schoolbook-mult	2.3	1.5	1.2	1.1
Schoolbook-shift	1.8	1.2	1.1	1.04
Karatsuba-mult	2.2	1.3	0.99	0.98
Karatsuba-shift	1.7	1.14	0.98	0.94

Table 4. Number of cycles for Horner's Rule based Hash(128-bit) for 50 Superblocks :with different strategies of multiplication and different levels of decimation

domly chosen independent keys for the two levels be (τ_1, τ_2), where each of them is a n-bit element of $GF(2^n)$.

If $\mathsf{len}(M)$, i.e.$B' \times 8$ is not a multiple of n, we pad the message with as many zeros as required to make it so. Let this padding function applied on the message M be denoted by the notation $\mathsf{pad}_n(M)$, which either returns the original message as it is if padding is not required or returns the appropriately padded message if padding is required. Let the length of the padded message be B bytes.

Now, we divide this padded message from the beginning into superblocks each containing 31 blocks. The last superblock may or may not contain 31 blocks, depending on the length of the input message. Now, for each superblock, the BRW polynomial based hashing is applied and the digest is calculated. As a result, we get a sequence of n-bit blocks. If the number of bits in the padded message is not more than $30 \times n$, then only the routine for partial superblock(1 to 30 blocks) BRW evaluation is applied on the message. If it is more than that, then on the initial complete superblocks, the BRW hash evaluation routine for complete superblock is applied and on the last superblock, if it is partial, the routine for BRW hash evaluation for partial block is applied. Now, the initial length of the input message (before padding), in number of bits, is expressed as a n-bit quantity. So, we get $\mathsf{bin}_n(\mathsf{len}(M))$ and it is appended to the digest sequence. Then on this sequence, the Horner's rule based hashing is applied. The result obtained is the final digest.

Let us define the following notation for this new hash function:
If M denotes the input message and (τ_1, τ_2) denotes the two randomly chosen independent keys, then the digest is denoted by:

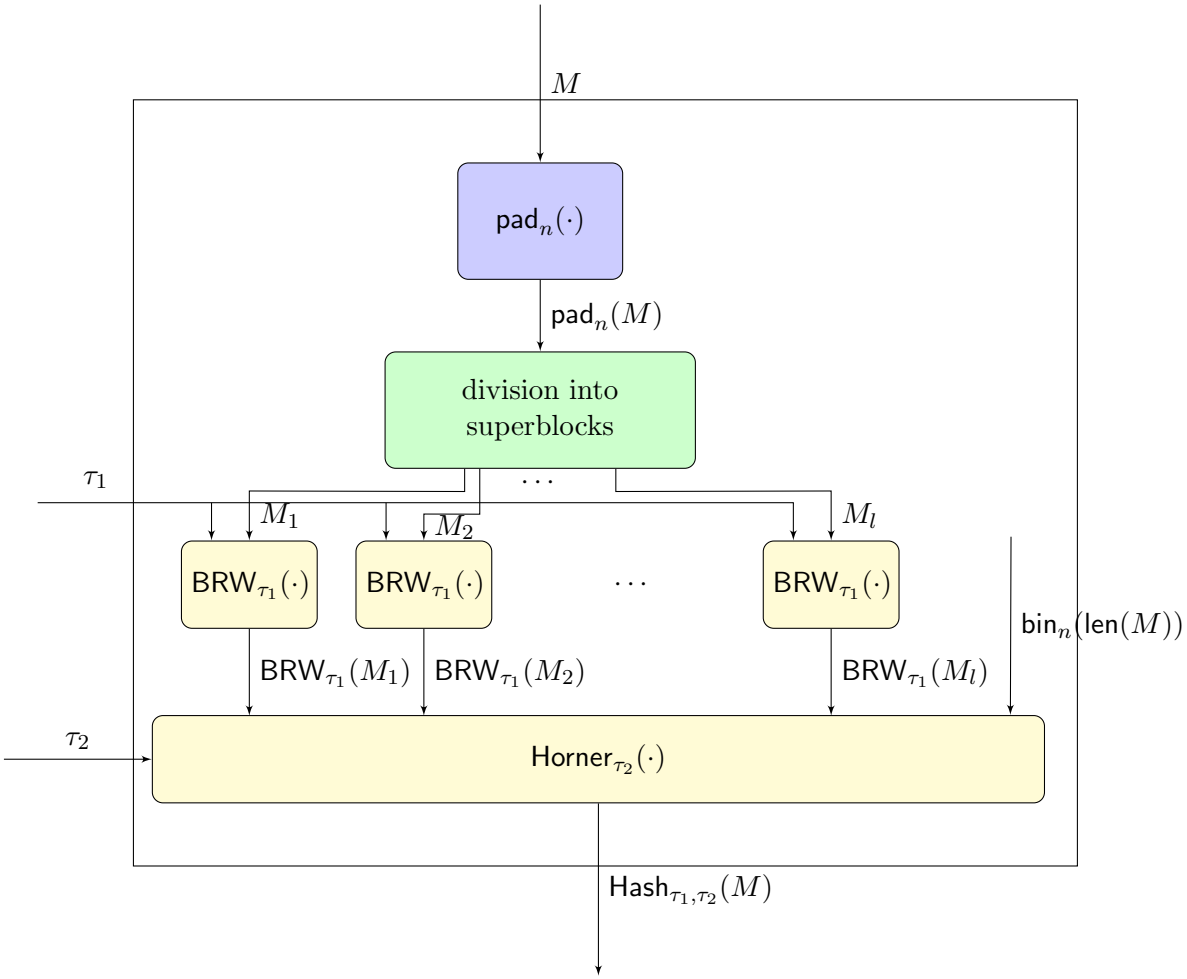
$$\mathsf{Hash}_{\tau_1,\tau_2}(M)$$

The pseudo-code for the complete hash function is as follows:

Algorithm 1 Evaluating $\mathsf{Hash}_{(\cdot,\cdot)}(\cdot)$

Input Input message M and the randomly chosen independent keys (τ_1, τ_2)
Compute $\mathsf{pad}_n(M)$
Let $l = \lceil \frac{\mathsf{len}(\mathsf{pad}_n(M))}{n \times 31} \rceil$
Divide the bit-representation of $\mathsf{pad}_n(M)$ into l superblocks.
Let the corresponding representation be $(M_1 || M_2 || \cdots || M_l)$
for i `from 1 to` l **do**
  Compute $\mathsf{BRW}_{\tau_1}(M_i)$
Output: $\mathsf{Hash}_{\tau_1,\tau_2}(M) = \mathsf{Horner}_{\tau_2}(\langle \mathsf{BRW}_{\tau_1}(M_1)) || \cdots || \mathsf{BRW}_{\tau_1}(M_l) || (\mathsf{bin}_n(\mathsf{len}(M))) \rangle)$

The design of the hash function can be explained with the following flow-chart:



As mentioned in the very beginning our target was to get a vector-input hash function with provably low collision and differential probabilities. We did not yet extend this hash function to a vector-input one, which we target to do in future.

But as a single input hash function, that it has provably low collision and differential probability can be proved.

Proposition 1. *Suppose the collision probability of* $\{\mathsf{BRW}_{\tau_1}\}_{\tau_1}$ *and* $\{\mathsf{Horner}_{\tau_2}\}_{\tau_2}$ *for equal length strings are at most* ϵ_1 *and* ϵ_2 *respectively. Suppose their differential probabilities for equal length strings are at most* ϵ_3 *and* ϵ_4 *respectively. (Note: As mentioned earlier these* ϵ_i*'s depend on length of the input. We consider that these particular values are corresponding to the length of the message, used in this proof.)*
Let $M \neq M'$ *be two input messages and our hash function is* $\{\mathsf{Hash}_{\tau_1,\tau_2}\}_{\tau_1,\tau_2}$*. Then*

$$Pr[\mathsf{Hash}_{\tau_1,\tau_2}(M) = \mathsf{Hash}_{\tau_1,\tau_2}(M')] \leq \epsilon_5$$

and for every α *in the underlying field*

$$Pr[\mathsf{Hash}_{\tau_1,\tau_2}(M) \oplus \mathsf{Hash}_{\tau_1,\tau_2}(M') = \alpha] \leq \epsilon_6.$$

Here the probabilities are taken over uniform random choice of τ_1, τ_2*.*

Proof. Let C and C' be the input bit-strings to the second level of our hash function,i.e. the Horner's rule based hash, corresponding to M and M' respectively. Then,

$$C = \langle \mathsf{BRW}_{\tau_1}(M_1)||\mathsf{BRW}_{\tau_1}(M_2)||\cdots||\mathsf{BRW}_{\tau_1}(M_s)||\mathsf{bin}_n(\mathsf{len}(M))\rangle$$
$$C' = \langle \mathsf{BRW}_{\tau_1}(M'_1)||\mathsf{BRW}_{\tau_1}(M'_2)||\cdots||\mathsf{BRW}_{\tau_1}(M'_t)||\mathsf{bin}_n(\mathsf{len}(M'))\rangle$$

where M_is and M'_is are superblocks, as defined earlier. It should be noted that, M_s and M'_t may be incomplete superblocks and may have trailing zeros due to padding.

Now, for the *first* part of the proposition let us consider two cases:

– Case 1:When the input messages are of different lengths:
In this case s and t may or may not be equal and $\mathsf{bin}_n(\mathsf{len}(M))$ and $\mathsf{bin}_n(\mathsf{len}(M'))$ parts of C and C' are certainly different. Hence,if s and t are equal,

$$\begin{aligned}
&Pr[\mathsf{Hash}_{\tau_1,\tau_2}(M) = \mathsf{Hash}_{\tau_1,\tau_2}(M')] \\
&= Pr[\mathsf{Horner}_{\tau_2}(C) = \mathsf{Horner}_{\tau_2}(C')] \\
&= Pr[\mathsf{Horner}_{\tau_2}(\langle \mathsf{BRW}_{\tau_1}(M_1)||\mathsf{BRW}_{\tau_1}(M_2)||\cdots||\mathsf{BRW}_{\tau_1}(M_s)||\mathsf{bin}_n(\mathsf{len}(M))\rangle) \\
&\quad = \mathsf{Horner}_{\tau_2}(\langle \mathsf{BRW}_{\tau_1}(M'_1)||\mathsf{BRW}_{\tau_1}(M'_2)||\cdots||\mathsf{BRW}_{\tau_1}(M'_s)||\mathsf{bin}_n(\mathsf{len}(M'))\rangle)] \\
&\leq \epsilon_2 \text{ from collision probability of Horner's rule based hash, since } C \text{ and } C' \text{ are different messages of same length.}
\end{aligned}$$

If s and t are not equal, without loss of generality, assume that $s > t$. In this case,

$$
\begin{aligned}
&Pr[\mathsf{Hash}_{\tau_1,\tau_2}(M) = \mathsf{Hash}_{\tau_1,\tau_2}(M')] \\
&= Pr[\mathsf{Horner}_{\tau_2}(C) = \mathsf{Horner}_{\tau_2}(C')] \\
&= Pr[\mathsf{Horner}_{\tau_2}(\langle \mathsf{BRW}_{\tau_1}(M_1)||\mathsf{BRW}_{\tau_1}(M_2)||\cdots||\mathsf{BRW}_{\tau_1}(M_s)||\mathsf{bin}_n(\mathsf{len}(M))\rangle) \\
&\quad = \mathsf{Horner}_{\tau_2}(\langle \mathsf{BRW}_{\tau_1}(M'_1)||\mathsf{BRW}_{\tau_1}(M'_2)||\cdots||\mathsf{BRW}_{\tau_1}(M'_t)||\mathsf{bin}_n(\mathsf{len}(M'))\rangle)] \\
&= Pr[\mathsf{Horner}_{\tau_2}(\langle \mathsf{BRW}_{\tau_1}(M_1)||\mathsf{BRW}_{\tau_1}(M_2)||\cdots||\mathsf{BRW}_{\tau_1}(M_s)||\mathsf{bin}_n(\mathsf{len}(M))\rangle) \\
&\oplus \mathsf{Horner}_{\tau_2}(\langle \mathsf{BRW}_{\tau_1}(M'_1)||\mathsf{BRW}_{\tau_1}(M'_2)||\cdots||\mathsf{BRW}_{\tau_1}(M'_t)||\mathsf{bin}_n(\mathsf{len}(M'))\rangle) = 0] \\
&\leq s/\sharp\mathbb{F}, \text{ if } \mathbb{F} \text{ is the underlying field.}
\end{aligned}
$$

The reason is that the left hand side of the expression inside the probability calculation is a non-zero polynomial in τ_2 of degree at most s over $\mathbb{F}$. It is non-zero, because the constant terms of the polynomials, i.e. $\mathsf{bin}_n(\mathsf{len}(M))$ and $\mathsf{bin}_n(\mathsf{len}(M'))$ are certainly different. So, it has at most s roots in $\mathbb{F}$. The polynomial is zero only if τ_2 takes one of these s values.

– Case 2: When the input messages are of same length:
Here, certainly $s = t$ and $\mathsf{len}(M) = \mathsf{len}(M')$. So,

$$
\begin{aligned}
&Pr[\mathsf{Hash}_{\tau_1,\tau_2}(M) = \mathsf{Hash}_{\tau_1,\tau_2}(M')] \\
&= Pr[\mathsf{Horner}_{\tau_2}(C) = \mathsf{Horner}_{\tau_2}(C')] \\
&= Pr[\mathsf{Horner}_{\tau_2}(C) = \mathsf{Horner}_{\tau_2}(C')|C = C']Pr[C = C'] \\
&\quad + Pr[\mathsf{Horner}_{\tau_2}(C) = \mathsf{Horner}_{\tau_2}(C')|C \neq C']Pr[C \neq C'] \\
&\leq 1.Pr[C = C'] + \epsilon_2, \text{ as } Pr[C \neq C'] \leq 1
\end{aligned} \tag{1}
$$

Now, let us find out $Pr[C = C']$.
We know,

$$
\begin{aligned}
C &= \langle \mathsf{BRW}_{\tau_1}(M_1)||\mathsf{BRW}_{\tau_1}(M_2)||\cdots||\mathsf{BRW}_{\tau_1}(M_s)||\mathsf{bin}_n(\mathsf{len}(M))\rangle \\
C' &= \langle \mathsf{BRW}_{\tau_1}(M'_1)||\mathsf{BRW}_{\tau_1}(M'_2)||\cdots||\mathsf{BRW}_{\tau_1}(M'_s)||\mathsf{bin}_n(\mathsf{len}(M'))\rangle
\end{aligned}
$$

As $M \neq M'$, there must exist atleast one i in the range $[1, s]$, such that $M_i \neq M'_i$.
Hence,

$$
\begin{aligned}
&Pr[C = C'] \\
&= Pr[\langle \mathsf{BRW}_{\tau_1}(M_1)||\mathsf{BRW}_{\tau_1}(M_2)||\cdots||\mathsf{BRW}_{\tau_1}(M_s)\rangle \\
&\quad = \langle \mathsf{BRW}_{\tau_1}(M'_1)||\mathsf{BRW}_{\tau_1}(M'_2)||\cdots||\mathsf{BRW}_{\tau_1}(M'_s)\rangle] \\
&\leq Pr(\mathsf{BRW}_{\tau_1}(M_i) = \mathsf{BRW}_{\tau_1}(M'_i)) \\
&= \epsilon_1, \text{ from collision probability assumption of BRW.}
\end{aligned}
$$

Hence, from (1),

$$\begin{aligned}
&Pr[\mathsf{Hash}_{\tau_1,\tau_2}(M) = \mathsf{Hash}_{\tau_1,\tau_2}(M')] \\
&= 1.Pr[C = C'] + \epsilon_2 \\
&\leq \epsilon_1 + \epsilon_2 \\
&= \epsilon_5 \text{ (say), which proves the first part of the proposition.}
\end{aligned}$$

Now, for the second part of the proposition, let us again consider similar cases: Here let α be an arbitrary element in the field.

– Case 1:When the input messages are of different lengths:
In this case s and t may or may not be equal and $\mathsf{bin}_n(\mathsf{len}(M))$ and $\mathsf{bin}_n(\mathsf{len}(M'))$ parts of C and C' are certainly different. Without loss of generality, assume that $s \geq t$. Hence,

$$\begin{aligned}
&Pr[\mathsf{Hash}_{\tau_1,\tau_2}(M) \oplus \mathsf{Hash}_{\tau_1,\tau_2}(M') = \alpha] \\
&= Pr[\mathsf{Horner}_{\tau_2}(C) \oplus \mathsf{Horner}_{\tau_2}(C') = \alpha] \\
&= Pr[\mathsf{Horner}_{\tau_2}(\langle \mathsf{BRW}_{\tau_1}(M_1)||\mathsf{BRW}_{\tau_1}(M_2)||\cdots||\mathsf{BRW}_{\tau_1}(M_s)||\mathsf{bin}_n(\mathsf{len}(M))\rangle) \\
&\quad \oplus \mathsf{Horner}_{\tau_2}(\langle \mathsf{BRW}_{\tau_1}(M'_1)||\mathsf{BRW}_{\tau_1}(M'_2)||\cdots||\mathsf{BRW}_{\tau_1}(M'_t)||\mathsf{bin}_n(\mathsf{len}(M'))\rangle) = \alpha] \\
&= Pr[\mathsf{Horner}_{\tau_2}(\langle \mathsf{BRW}_{\tau_1}(M_1)||\mathsf{BRW}_{\tau_1}(M_2)||\cdots||\mathsf{BRW}_{\tau_1}(M_s)||\mathsf{bin}_n(\mathsf{len}(M))\rangle) \\
&\oplus \mathsf{Horner}_{\tau_2}(\langle \mathsf{BRW}_{\tau_1}(M'_1)||\mathsf{BRW}_{\tau_1}(M'_2)||\cdots||\mathsf{BRW}_{\tau_1}(M'_t)||\mathsf{bin}_n(\mathsf{len}(M'))\rangle) \oplus \alpha = 0] \quad (2)
\end{aligned}$$

Now the left hand side of the expression inside probability of (2) is a polynomial whose constant term is $\alpha \oplus \mathsf{bin}_n(\mathsf{len}(M)) \oplus \mathsf{bin}_n(\mathsf{len}(M'))$, which may vanish depending on the value of these elements. So, the previous type of argument fails here.

Hence, to make the differential probability of the designed hash function low, we need an extra multiplication by the key at the Horner level.
So, our function for the Horner's rule is redefined as follows:
$\mathsf{Horner}_x(f_d, \cdots, f_0) = ((((f_d x + f_{d-1})x + f_{d-2})x + \cdots + f_1)x + f_0)x$, where f_i's, for $0 \leq i \leq d$, are n-bit blocks as usual.
In this case the left hand side of the expression inside probability of (2) is a polynomial of degree at most $s+1$ and with constant term only α. Moreover, as the lengths of C and C' are different, this is definitely a non-zero polynomial.
Hence, by similar arguments as in first part of the proposition, the differential probability in this case is bounded by $\frac{s+1}{\sharp\mathbb{F}}$.

– Case 2: When the input messages are of same length:
Here, certainly $s = t$ and $\mathsf{len}(M) = \mathsf{len}(M')$. So,

$Pr[\mathsf{Hash}_{\tau_1,\tau_2}(M) \oplus \mathsf{Hash}_{\tau_1,\tau_2}(M') = \alpha]$
$= Pr[\mathsf{Horner}_{\tau_2}(C) \oplus \mathsf{Horner}_{\tau_2}(C') = \alpha]$
$= Pr[\mathsf{Horner}_{\tau_2}(C) \oplus \mathsf{Horner}_{\tau_2}(C') \oplus \alpha = 0 | C = C'] Pr[C = C']$
$\quad + Pr[\mathsf{Horner}_{\tau_2}(C) \oplus \mathsf{Horner}_{\tau_2}(C') \oplus \alpha = 0 | C \neq C'] Pr[C \neq C']$
$\leq 1.\epsilon_1 + Pr[\mathsf{Horner}_{\tau_2}(C) \oplus \mathsf{Horner}_{\tau_2}(C') \oplus \alpha = 0 | C \neq C'].1$, putting values from first part.
$\leq \epsilon_1 + \epsilon_4$, from differential probability of Horner
$= \epsilon_6$(say), which completes the proof of the second part of the proposition.

Hence, the proposition is proved. □

Now, in each of the two levels of the hash function, we have four different choices due to batching or decimation. Any choice for the first level can be integrated with any choice for the second level. Very reasonably in most of the cases, we integrated a particular level of batching for BRW with the same level of decimation for Horner, as level of batching/decimation is related to speed proportionally. We also tried to integrate third level of batching of BRW with fourth level of decimation of Horner, as the structure of BRW in level three batching is nice and it does not mix with other levels of batching. Therefore, we expected a competitive result from third level of pipelining for BRW than others and the timing result mostly supports that expectation. Again, as usual Horner at fourth decimation level was expected to be the best among all decimation levels. Hence, we integrated these two also. The comparative study of speed of the total Hash function is as follows:

Strategy+Batching level	Number of Bytes					
	1	10	100	1000	10000	30000
Schoolbook-mult + 1	4.04	0.4	1.28	0.7	0.6	0.57
Schoolbook-mult + 2	4.53	0.45	1.25	0.68	0.55	0.55
Schoolbook-mult + 3	4.53	0.45	1.24	0.68	0.53	0.53
Schoolbook-mult + 4	4.55	0.45	1.29	0.69	0.53	0.52
Schoolbook-mult + 5	4.55	0.45	1.29	0.67	0.53	0.52
Schoolbook-shift + 1	4.04	0.4	1.14	0.68	0.58	0.58
Schoolbook-shift + 2	4.56	0.46	1.09	0.65	0.56	0.56
Schoolbook-shift + 3	4.53	0.45	1.1	0.66	0.54	0.53
Schoolbook-shift + 4	4.53	0.45	1.15	0.66	0.54	0.52
Schoolbook-shift + 5	4.53	0.45	1.15	0.65	0.53	0.52
Karatsuba-mult + 1	3.65	0.36	1.32	0.74	0.6	0.59
Karatsuba-mult + 2	4.6	0.45	1.2	0.72	0.6	0.57
Karatsuba-mult + 3	4.5	0.45	1.29	0.69	0.54	0.53
Karatsuba-mult + 4	4.5	0.45	1.31	0.71	0.54	0.53
Karatsuba-mult + 5	4.53	0.45	1.31	0.7	0.54	0.53
Karatsuba-shift + 1	4.05	0.4	1.15	0.71	0.59	0.59
Karatsuba-shift + 2	4.53	0.45	1.06	0.69	0.58	0.57
Karatsuba-shift + 3	4.53	.45	1.12	0.65	0.54	0.53
Karatsuba-shift + 4	4.53	0.45	1.15	0.66	0.53	0.52
Karatsuba-shift + 5	4.54	0.45	1.15	0.66	0.53	0.53

Table 5. Speed of Hash (in number of cycles per byte): with different strategies of multiplication and different levels of batching

4.5 Theoretical comparison between the new hash function and conventional Horner's rule based hash function:

The major time consuming operation in both the conventional Horner's rule based hash and the new hash function is multiplication of field elements. Let us compare between the number of multiplications required for same length message in the two schemes.

Let M be the input message. For simplicity of notation let us assume that $\mathsf{len}(M)$ is multiple of n. Hence, number of blocks in the message is $\frac{\mathsf{len}(M)}{n}$.
Number of superblocks in the message is $\lceil \frac{\mathsf{len}(M)}{n \times 31} \rceil = k$ (say).

So, the number of field multiplications required in Horner's rule based hash is clearly one less than the number of blocks in the message, i.e.

$$\frac{\mathsf{len}(M)}{n} - 1$$

If we now consider the new hash function, in the first level, i.e. in the BRW level computing the hash value for each superblock requires *fifteen* multiplications, possibly except the last superblock. Again, for simplicity, we ignore this fact about last superblock and take the number of multiplications required for it also to be fifteen.
So, the total number of multiplications in this level is $15 \times k$.
In the second level there are $k+1$ blocks, k digests from k superblocks along with the block for length padding and as ususal this requires k multiplications.

So, the total number of field multiplications required in the new hash function is

$$16 \times k = 16 \times \lceil \frac{\mathsf{len}(M)}{n \times 31} \rceil \approx 0.5 \times \lceil \frac{\mathsf{len}(M)}{n} \rceil,$$

which is almost 0.5 times the number of multiplications required for the Horner's rule based hash.

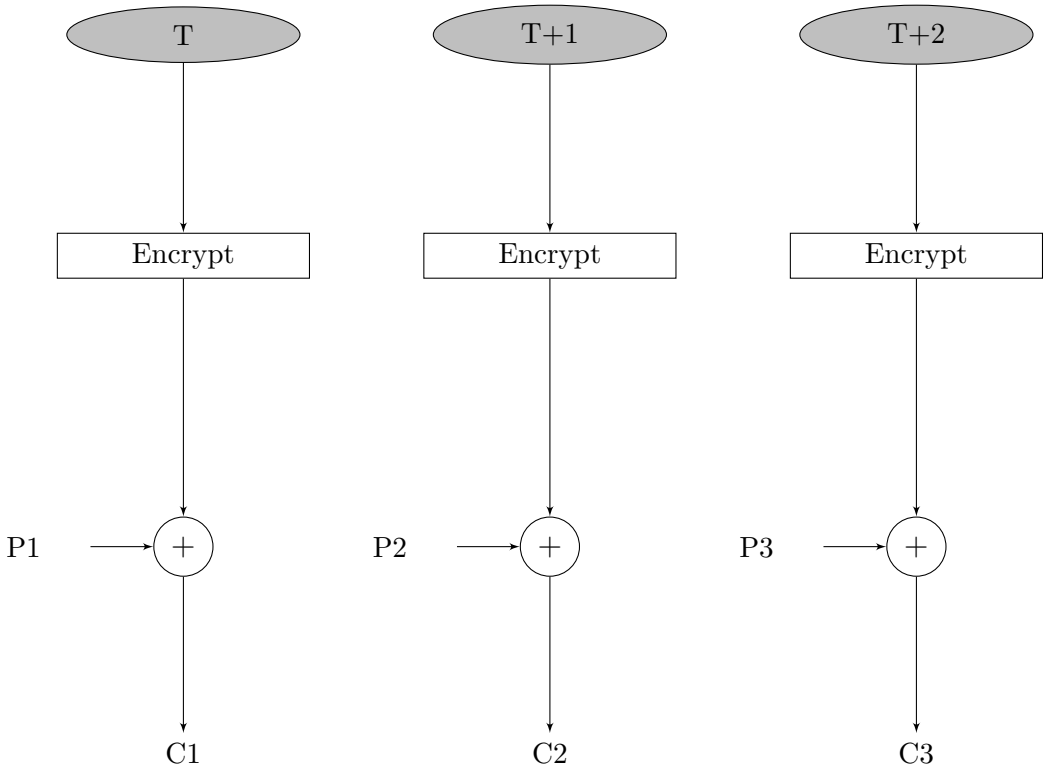
5 Integration of Hash function with AES to obtain some encryption and authentication primitives:

In [1] different encryption and authentication primitives have been proposed. All of these require one stream cipher supporting an IV along with a suitable hash function. A Block cipher in counter mode can also be used in place of a stream cipher along with some minor modifications to achieve the same level of security.

5.1 Counter mode:

Counter mode is one of the well known modes of operation for a block cipher. Let, the plaintext or the message to be encrypted be denoted by M. In counter mode, we choose a counter, which is a bitstring of length $\mathsf{len}(M)$. This counter is encrypted with the key. The encrypted result is xored with the plaintext to produce the ciphertext. Counter for each plaintext will be different. We can use as a counter any function which produces a sequence which is guaranteed not to repeat for a long time. For PRF security, this counter is selected in the following way: a bitstring of length $\mathsf{len}(M)$ is randomly chosen, which is termed as nonce or IV. If E_k denotes the block cipher with k as key, then the counter is set as, $ctr = E_k(IV)$. This nonce, generally, is not a secret quantity.

Let T denotes the counter, ctr derived in that way and C_i denotes the ciphertext corresponding to plaintext P_i then the counter mode encryption can be described as:
$O_i = E_k(i)$
$C_i = P_i \oplus O_i$



Here, we have used here Advanced Encryption Standard (AES) in counter mode in place of the stream cipher and the hash function described above in place of the hash function to implement some of the primitives described in the main paper. For this work, AES in counter mode, along with its different modules, have been implemented by Dr. Chakroborty. Among some of the primitives designed in the paper [1] and implemented here let us take one example of a MAC scheme and another example of an AE scheme:

$\mathbf{SC - MACa_K(M_1)}$

1.$\tau = SC_K(fStr)$;

2.$N = Hash_\tau(M_1)$;

3.$tag = SC_K(N)$;

return tag.

fStr is a fixed string, which is generally a parameter of the specific algorithm. We have fixed it to all-zero string. For implementing this scheme, we have at first expanded the input key, as is natural in AES, just once. Reasonably enough, this is not included in timing measurement. Then wherever, more than one block (n-bit string) is output of stream cipher application in the above scheme, we use AES in counter mode in place of it. Only when output of some stream cipher application is one block, we use normal AES encryption

in its place. In place of Hash, we have used our hash function. The speed achived is as follows:

Strategy+Batching level	Number of Bytes					
	1	10	100	1000	10000	30000
Schoolbook-mult + 1	106.785	10.65	3.03	0.82	0.58	0.57
Schoolbook-mult + 2	108.927	10.69	3.03	0.82	0.56	0.55
Schoolbook-mult + 3	106.797	10.68	3.03	0.805	0.54	0.52
Schoolbook-mult + 4	107.115	10.71	3.03	0.84	0.56	0.53
Schoolbook-mult + 5	107.1	10.71	3.03	0.82	0.56	0.54
Schoolbook-shift + 1	106.572	10.65	2.88	0.82	0.59	0.58
Schoolbook-shift + 2	107.022	10.7	2.88	0.81	0.57	0.55
Schoolbook-shift + 3	106.803	10.68	2.89	0.8	0.54	0.53
Schoolbook-shift + 4	107.112	10.71	10.89	0.81	0.56	0.54
Schoolbook-shift + 5	107.115	10.71	2.89	0.799	0.54	0.52
Karatsuba-mult + 1	106.575	10.65	3.02	0.85	0.62	0.6
Karatsuba-mult + 2	106.938	10.69	3.02	0.86	0.59	0.58
Karatsuba-mult + 3	106.812	10.69	2.99	0.81	0.56	0.54
Karatsuba-mult + 4	107.109	10.71	2.99	0.83	0.57	0.55
Karatsuba-mult + 5	107.097	10.94	2.99	0.83	0.57	0.54
Karatsuba-shift + 1	106.581	10.75	2.84	0.85	0.6	0.59
Karatsuba-shift + 2	106.971	10.69	2.84	0.83	0.59	0.57
Karatsuba-shift + 3	106.782	10.67	2.83	0.79	0.54	0.53
Karatsuba-shift + 4	107.31	10.73	2.85	0.8	0.54	0.53
Karatsuba-shift + 5	107.313	10.73	2.85	0.792	0.54	0.52

Table 6. Speed of SC-MACa (in number of cycles per byte for 128-bit): with different strategies of multiplication and different levels of batching

$\mathbf{AE-2a.Encrypt_K(N,M)}$

1.$K' = SC_K(fStr)$;

2.$\tau = SC_K(K')$;

3.$(R, Z) = SC_K(N)$;

4.$C = M \oplus Z$;

5.$tag = Hash_\tau(C \oplus R)$;

return (C, tag).

Here also fStr is set to all-zero string and N is nonce. Similar implementation strategy as the above MAC scheme is also taken here and the speed result is as follows:

Strategy+Batching level	Number of Bytes					
	1	10	100	1000	10000	30000
Schoolbook-mult + 1	288.147	28.82	3.35	1.5	1.28	1.34
Schoolbook-mult + 2	290.259	29.02	3.36	1.5	1.25	1.31
Schoolbook-mult + 3	290.26	28.98	3.35	1.48	1.23	1.29
Schoolbook-mult + 4	287.304	28.84	3.34	1.5	1.23	1.29
Schoolbook-mult + 5	289.038	28.90	3.35	1.49	1.25	1.30
Schoolbook-shift + 1	288.273	28.82	3.2	1.5	1.28	1.34
Schoolbook-shift + 2	290.265	28.98	3.21	1.48	1.25	1.31
Schoolbook-shift + 3	290.262	29.02	3.20	1.46	1.23	1.29
Schoolbook-shift + 4	289.044	28.67	3.21	1.48	1.24	1.31
Schoolbook-shift + 5	288.135	28.81	3.22	1.47	1.23	1.29
Karatsuba-mult + 1	288.024	28.80	3.34	1.54	1.3	1.36
Karatsuba-mult + 2	287.289	28.78	3.32	1.5	1.2	1.35
Karatsuba-mult + 3	289.821	29.02	3.33	1.5	1.25	1.31
Karatsuba-mult + 4	289.041	28.904	3.31	1.52	1.25	1.31
Karatsuba-mult + 5	289.041	28.904	3.31	1.47	1.25	1.31
Karatsuba-shift + 1	288.267	28.82	3.21	1.49	1.30	1.35
Karatsuba-shift + 2	289.821	28.98	3.19	1.477	1.29	1.34
Karatsuba-shift + 3	289.818	28.98	3.20	1.47	1.23	1.29
Karatsuba-shift + 4	288.144	28.92	3.19	1.481	1.24	1.29
Karatsuba-shift + 5	288.792	28.88	3.19	1.468	1.23	1.28

Table 7. Speed of AE-2a (in number of cycles per byte for 128-bit): with different strategies of multiplication and different levels of batching

6 Integration of Hash function with Salsa20 to obtain some encryption and authentication primitives:

One implementation of Salsa20, obtained from [6], has also been used in place of the stream cipher to implement some of the primitives described in the main paper. This particular implementation of Salsa20 is available in assembly language and uses SSE2 instructions. Along with it, all necessary supporting header files are also taken from the same source. Now, we integrate this implementation along with our hash function implementation to implement some of the basic primitives. One of the primitives mentioned above is implemented in this way also and the scheme is once again repeated here for ease of referencing.

$\mathbf{AE-2a.Encrypt_K(N,M)}$

$1.K' = SC_K(fStr);$

$2.\tau = SC_K(K');$

$3.(R,Z) = SC_K(N);$

$4.C = M \oplus Z;$

$5.tag = Hash_\tau(C \oplus R);$

return (C, tag).

In the implementation, key setup is done just once and is not included in timing measurement, as is natural for any stream cipher application. Each time the stream cipher is invoked, the IV is set up. In place of Hash, the hash function we implemented has been used.

This is just a basic implementation of ae2a using salsa20. Speed achieved has not yet been taken as of main concern here. The target was to get just a working example of integration of Salsa20 with our hash function. That target is achieved. After this we may modify some input techniques to improve the speed achieved and we target for the same.

7 Conclusion

So far the most significant and fundamental part of this work is the design of the hash function, which seems to be quite competitive in terms of speed with the other hash functions available so far. Its utility has been shown by integrating with AES or Salsa20 to get the implementation of the primitives mentioned in [1]. In that sense, it gives a practical form to this primitives defined only theoretically so far. Though it seems that the integration of this hash function with AES could be done reasonably here, the integration with Salsa20 needs to be done more carefully and then only proper comparison of speeds achieved from these two schemes can be done. That forms the immediate scope of improvement for this work.

Moreover, this work of implementing some encryption and authentication schemes by integrating the hash function with AES or Salsa20 has so far been done only when $n = 128$, i.e. when the underlying field is GF(2^{128}). The same work must be done for $n = 256$, i.e. for the field GF(2^{256}).

8 acknowledgement

I would like to express my deepest gratitude to my advisor Prof. Palash Sarkar for his guidance and support. His extreme energy, creativity and excellent mentoring skills have

always been a constant source of motivation for me. The perfection that he brings to each and every piece of work that he does always inspired me at each step of my work. He is a great person and one of the best mentors. I am always thankful to him.
I would also like to thank Dr. Debrup Chakraborty for devoting his time in discussing some useful design and implementation ideas with me.

References

1. P. Sarkar, *Modes of operations for encryption and authentication using stream ciphers supporting an initialisation vector*, Cryptography and Communications, 2014.
2. `http://cr.yp.to/snuffle.html`
3. D.J. Bernstein, *Polynomial Evaluation and Message Authentication*, `http://cr.yp.topapers.html#pema`, 2011.
4. P. Sarkar, *Efficient tweakable enciphering schemes from (block-wise) universal hash functions*, IEEE Transactions on Information Theory, 2009
5. Gueron, Shay, and Michael E. Kounavis. *Intel Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode*, Intel white paper (September 2012) (2010).
6. `http://cr.yp.to/snuffle.html`
7. J. Daemen and V. Rijmen, *The Design of Rijndael:AES - The Advanced Encryption Standard*, Springer-Verlag, 2002.
8. D. R. Stinson, *Cryptography: Theory and Practice*, CRC Press LLC.
9. D. Chakraborty, C. Mancillas-Lopez, F. Rodriguez-Henriquez, P. Sarkar. *Efficient hardware implementations of brw polynomials and tweakable enciphering schemes.* Computers, IEEE Transactions on, 62(2), 279-294, 2013.
10. L. Carter, M.N. Wegman. *Universal classes of hash functions.* J. Comput. Syst. Sci. 18(2), 143154, 1979.
11. S. Halevi, H. Krawczyk. *MMH: Software message authentication in the gbit/second rates.* Biham E. (ed.) Fast Software Encryption, volume 1267 of Lecture Notes in Computer Science, pp. 172189. Springer, 1997.
12. E.N. Gilbert, F.J. MacWilliams, N.J.A. Sloane. *Codes which detect deception* Bell Syst. Tech. J. 53, 405424, 1974.
13. J. Black, S. Halevi, H. Krawczyk, T. Krovetz, P. Rogaway. *UMAC: fast and secure message authentication.* Wiener M.J. (ed.) CRYPTO, volume 1666 of Lecture Notes in Computer Science, pp. 216233. Springer, 1999.
14. D.J. Bernstein. *The Poly1305-AES message-authentication code.* Gilbert H., Handschuh H. (eds.) FSE, volume 3557 of Lecture Notes in Computer Science, pp. 3249. Springer, 2005.
15. T. Krovetz, P. Rogaway. *Fast universal hashing with small keys and no preprocessing: the PolyR construction.* Won D. (ed.) ICISC, volume 2015 of Lecture Notes in Computer Science, pp. 7389. Springer, 2000.
16. T. Johansson. *Bucket hashing with a small key size.* EUROCRYPT, pp. 149162, 1997.
17. P. Rogaway. *Bucket hashing and its application to fast message authentication.* J. Cryptol. 12(2), 91115, 1999.
18. V. Shoup. *On fast and provably secure message authentication based on universal hashing.* N. Koblitz(ed.) CRYPTO, volume 1109 of Lecture Notes in Computer Science, pp. 313328. Springer, 1996.
19. T. Johansson. *A shift register construction of unconditionally secure authentication codes.* Des. Codes Cryptogr. 4(1), 6981, 1994.
20. G. Kabatianskii, B.J.M. Smeets, T. Johansson. *On the cardinality of systematic authentication codes via error-correcting codes.* IEEE Trans. Inf. Theory 42(2), 566578, 1996.
21. H. Krawczyk. *LFSR-based hashing and authentication.* Desmedt (ed.) CRYPTO, volume 839 of Lecture Notes in Computer Science, pp. 129139. Springer, 1994.
22. D.R. Stinson. *Universal hashing and authentication codes.* Des. Codes Cryptogr. 4(4), 369380, 1994.
23. P. Sarkar. *A new multi-linear universal hash family.* Designs, codes and cryptography 69.3 : 351-367, 2013.
24. M. Nandi. *On the minimum number of multiplications necessary for universal hash functions.* Fast Software Encryption (pp. 489-508). Springer Berlin Heidelberg, 2014.

25. M. O. Rabin, S. Winograd. *Fast Evaluation of Polynomials by Rational Preparation.* Communications on Pure and Applied Mathematics XXV , 433458, 1972.
26. S. Winograd. *A new algorithm for inner product.* IEEE Transactions on Computers 17, 693694, 1968.
27. T. Krovetz. *Message authentication on 64-bit architectures.* Selected areas in cryptography, Lecture Notes in Computer Science, Springer Verlag 4356, 327341, 2007.
28. M. Boesgaard, O. Scavenius, T. Pedersen, T. Christensen, E. Zenner. *Badgera fast and provably secure MAC* Applied cryptography and network security, 176191, 2005.
29. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Amit Sahai. *Cryptography with Constant Computational Overhead.* STOC 2008, 433442, 2008.
30. D. A. McGrew, J. Viega. *The security and performance of the Galois/counter mode of operation.* URL: http://eprint.iacr.org/2004/193/, 2004.
31. J. Bierbrauer, T. Johansson, G. Kabatianskii, B. J. M. Smeets. *On families of hash functions via geometric codes and concatenation.* Advances in Cryptology Crypto, Lecture Notes in Computer Science, Springer-Verlag 773, 331342, 1993.
32. B. den Boer. *A simple and key-economical unconditional authentication scheme.* Journal of Computer Security 2, 6571, 1993.
33. Richard Taylor. *An integrity check value algorithm for stream ciphers.* Advances in Cryptology Crypto, Lecture Notes in Computer Science, Springer-Verlag (1993), 4048, 1993.
34. T. Krovetz, and P. Rogaway. *The software performance of authenticated-encryption modes.* Fast Software Encryption, pp. 306-327. Springer Berlin Heidelberg, 2011.
35. `http://web.cs.ucdavis.edu/~rogaway/ocb/news/code/timing_x86.c`
36. S. Gueron, M. E. Kounavis. *Intel(R) Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode.* White Paper, 2010.

M.TECH(CS) DISSERTATION THESIS COMPLETION CERTIFICATE

Student : Sebati Ghosh (MTC1320)
Topic : A New Universal Hash Function: Its Integration with Block and Stream Ciphers for Encryption and Authentication
Supervisor : Prof. Palash Sarkar

This is to certify that the thesis titled "A New Universal Hash Function: Its Integration with Block and Stream Ciphers for Encryption and Authentication" submitted by Sebati Ghosh in partial fulfillment for the award of the degree of Master of Technology is a bonafide record of work carried out by her under my supervision. The thesis has fulfilled all the requirements as per the regulations of this Institute and, in our opinion, has reached the standard needed for submission. The results embodied in this thesis have not been submitted to any other university for the award of any degree or diploma.

Palash Sarkar

Date : 10^{th} $July$**,** 2015