A High Speed Hardware Architecture for Universal Message Authentication Code

Bo Yang       Ramesh Karri

Department of Electrical and Computer Engineering

Polytechnic University, Brooklyn, NY, 11201

yangbo@photon.poly.edu, ramesh@india.poly.edu

David A. McGrew

Cisco Systems, Inc.

San Jose, CA 95134

mcgrew@cisco.com
Abstract

We present an architecture level optimization technique called divide-and-concatenate based on two observations: (i) the area of a multiplier and associated data path decreases quadratically and their speeds increase gradually as their operand size is reduced. (ii) in universal hash functions and associated message authentication codes, two functions are equivalent if they have the same collision probability property. In the proposed approach we divide a 2w-bit data path (with collision probability $2^{-2w}$) into two w-bit data paths (each with collision probability $2^{-w}$) and concatenate their results to construct an equivalent 2w-bit data path (with a collision probability $2^{-2w}$).

We applied this technique on NH universal hash, a universal hash function that uses multiplications and additions. When compared to the straightforward 32-bit pipelined NH universal hash data path, the divide-and-concatenate approach yields a 94% increase in throughput with only 40% hardware overhead. The NH universal hash associated message authentication code UMAC architecture with collision probability $2^{-32}$ that uses four equivalent 8-bit divide-and-concatenate NH universal hash data paths yields a throughput of 79.2 Gbps with only 3840 FPGA slices when implemented on a Xilinx Virtex II XC2VP7-7 Field Programmable Gate Array (FPGA) device.

**Keywords:** UMAC, Universal hash functions, Performance optimization, Divide-and-concatenate
1. Introduction

Message Authentication Codes (MACs) are used by a receiver to assure the integrity of a message. The receiver computes the MAC of the received message using the secret key shared with the sender. If the computed MAC does not match the received MAC, it shows that the message was modified by a third party during transmission [1] [2] [4].

Early MAC algorithms use symmetric key based block ciphers, such as Advanced Encryption Stand (AES), in the Cipher Block Chaining (CBC) mode [2] [4]. In the CBC mode, each message block is xored with the ciphertext output from the previous step before encrypting. This is repeated until all the message blocks are consumed. The resulting ciphertext is then used as the MAC. Security of the underlying block cipher assures the security of the CBC MAC. The second class of MACs uses cryptographic hash functions such as MD5 [5] and SHA-1 [6]. Since MD5 and SHA-1 do not use keying, HMAC was defined by combining them with secret keys. HMAC has a well-understood proof of strength based on reasonable assumptions concerning the underlying hash function [1] [8].

MAC has been widely used in many network applications. Secure Socket Layer (SSL) and Transport Layer Security (TLS) use encryption and MAC to support secure browsing, secure file transfer and secure remote login between end users and servers. IPsec tunnel mode is used to setup a Virtual Private Network (VPN) between two enterprise networks over Internet [9]. Routing protocols have begun to use message authentication to verify the routing information transferred among routers [10]. These high performance commercial servers and routers require dedicated
hardware implementations of MACs that match the greater than 10 Gbps wire speed requirements of network traffic.

However, CBC-MAC, MD5 and SHA-1 algorithm are iterative; the current computation step depends on the result of the previous step. They are not parallelizable, so implementing them in hardware only yields moderate throughput due to their iterative structure. FPGA implementations of several symmetric key algorithms have been evaluated on Xilinx Virtex devices [11]. The fastest RC6 in CBC mode generates 126.5 Mbps throughput consuming 3189 slices. The fastest AES in CBC mode generates 300.1 Mbps using 5302 slices. The faster Serpent in CBC mode generates 444.2 Mbps consuming 7964 slices. Twofish in CBC mode generates 122.1 Mbps consuming 4012 slices. Commercial implementations of MD5 and SHA-1 targeting Xilinx Virtex II have a throughput of about 744 Mbps and 874 Mbps respectively [12] [13].

Recently, universal hash functions such as NH [14], MMH [15], TMMH [16] and Square Hash [17] have been designed to exploit architectural support for multiplication in modern processors. They make use of one-cycle multiplication provided by underlying processors to speedup the hash calculation. They do not have iterative internal structure and are parallelizable. Even the software implementations can achieve greater than 1 Gbps throughput. NH universal hash based Universal Message Authentication Code (UMAC) with collision probability of $2^{-60}$ can achieve 2.9 Gbps when implemented in software [14].

In this paper we show that hardware implementations of UMAC that can achieve the greater than 10 Gbps throughput requirement of high performance servers and routers. We present an
architecture level optimization technique named divide-and-concatenate to speed up the hardware implementations of universal hash functions and associated MACs without compromising the collision probability by using several small size data paths to construct an equivalent large data path. [14]~[17] show that the collision probability of software implementations of universal hash functions can be improved even when the underlying processor architecture can not provide required collision probability. Because the width of data path is already determined by the hardware of the underlying processor, this technique cannot be used to improve the throughput of software implementations.

In the rest of this paper we will describe the divide-and-concatenate hardware architecture optimization technique for universal hash based MACs. Specifically, we will introduce universal hash functions and NH universal hash function in section 2. The motivation for divide-and-concatenate technique is presented in section 3. We will apply the divide-and-concatenate technique to NH universal hash in section 4. In section 5 we will present the architecture for the UMAC based on the NH universal hash architectures from section 2. We will summarize our contributions in section 6.

2. NH Universal Hash

Universal Message Authentication Code (UMAC) is a NESSIE (New European Schemes for Signatures, Integrity, and Encryption) message authentication code standard [18]. The core of UMAC is the NH universal hash function. In this section we will describe universal hash functions in general and the NH universal hash in particular.
2.1. Hash Function

A hash function ($h$) converts an input from a large domain ($x$) into an output in a small range (the hash value $y=h(x)$, often a subset of the integers). There are three features for a hash function:

**Pre-image resistance:** for a given $y$, it is infeasible to find an $x$ so that $h(x)$ is equal to $y$.

**Second pre-image resistance:** for a given $y$ that is equal to $h(x)$, it is infeasible to find a $x'$ so that $h(x')$ is equal to $h(x)$.

**Collision resistance:** it is infeasible to find a pair of $x$ and $x'$ so that $h(x)$ is equal to $h(x')$.

A hash function that meets the pre-image resistance and the second pre-image resistance is a one-way hash function [7]. Further, if a one-way hash function has collision resistance, it is collision resistant. Since collision-resistant hash functions are a subset of one-way hash functions, the collision probability, which stands for the ability of collision resistance, is the most important parameter of a hash function [7].

2.2. Universal Hash Function

Carter and Wegman [19] defined a universal hash function as follows: Let $A$ and $B$ be two sets, and let $H$ be a family of functions from $A$ to $B$. For every pair $x_1, x_2 \in A$ with $x_1 \neq x_2$, $h(x_1), h(x_2) \in B$ with $h \in H$, if the collision probability of $h(x_1) = h(x_2)$ equals to $1/|B|$, $H$ is a universal family of hash functions. $|B|$ is size of set $B$ and $1/|B|$ is the smallest possible value of the probability. When $B$ is small, the collision probability is large.

A MAC that uses a universal hash function as a building block hashes the input message $M$
down to a small-size hash value using the universal hash function and then applies a cryptographic primitive to this hash value [20]. Since universal hash functions can compress the message M efficiently, the associated MACs are fast. Several universal hash functions and associated MACs have been proposed including NH hash [14], MMH [15], TMMH [16] and Square Hash [17].

2.3. **NH Hash Algorithm**

NH universal hash is a universal hash function that uses additions and multiplications; the operations correspond to machine instructions on modern processors. When NH universal hash is implemented on a modern processor, it can calculate the hash value for a 1024 word (a word is 32-bit wide) message using 1024 32-bit word subkeys as follows:

$$\text{Hash Value} = (M_1 +_{32} K_1) \times (M_2 +_{32} K_2) +_{64} \ldots +_{64} (M_{i-1} +_{32} K_{i-1}) \times (M_i +_{32} K_i) \ldots +_{64} (M_{1023} +_{32} K_{1023}) \times (M_{1024} +_{32} K_{1024})$$

$M_i$ and $K_i$ are 32-bit message words and corresponding subkey words. $+_{32}$ and $+_{64}$ are addition mod $2^{32}$ and addition mod $2^{64}$ respectively.

3. **Motivation**

Cryptographic algorithms such as block encryption and message authentication are iterative, data-driven algorithms. These algorithms take an input message and a user key and generate a result after several iterations. Since these cryptographic algorithms are data-dominated, their hardware implementations are data path dominated with only a small amount of control logic. Arithmetic operations such as add, multiply and shift/rotate are at the core of these cryptographic algorithms.
One straightforward approach to speeding up cryptographic hardware is to use fast implementations of adders, multipliers and other components. Orthogonal to the circuit level and logic level approaches are architectural level speed-up techniques such as retiming [22] [25], pipelining [23] [24], loop unrolling [11] and parallel data path [26].

3.1. **Straightforward NH Universal Hash Data Path**

A 32-bit NH universal hash data path shown in Figure 1 that operates on 32-bit input message words and 32-bit subkey words has a collision probability of $2^{-32}$. A $w$-bit NH universal hash data path that operates on $w$-bit input message words and $w$-bit subkey words has a collision probability of $2^{-w}$ [14]. It is the core building block of Universal Message Authentication Code (UMAC) [14]. This data path can be implemented as a three-stage pipeline. In the first stage, two 32-bit adders are used to add two adjacent message words to their corresponding subkeys. In the second stage, these two 32-bit intermediate results are multiplied. In the final stage the 64-bit result from a multiplier is accumulated into an output register using a 64-bit adder. We implemented this three-stage pipeline on a Xilinx Virtex II XC2VP7-7 FPGA device using a single cycle carry-look-ahead adder and a single cycle array multiplier [27]. The adder and multiplier are generated by Xilinx core generator [28]. Since the maximum clock rate of a 64-bit combinational adder and the maximum clock rate of a 32-bit combinational array multiplier when they are implemented on the targeted Xilinx Virtex II FPGA is 193 MHz and 83 MHz respectively, the throughput of this design is limited by the throughput of the multiplier and equals 5.3 Gbps (=64 bits×83 MHz). Replacing the single cycle multiplier by a 5-stage 32-bit pipelined multiplier doubles the clock rate of the multiplier stage and
that of the design to 160 MHz. This in turn doubles the throughput of the design to 10.2 Gbps (=64 bits×160 MHz). Finally, we can replicate this pipelined data path, with each copy operating on an independent input stream to get additional improvement in throughput. However, this approach doubles the hardware cost.

![Diagram](image)

**Figure 1:** 32-bit data path for the NH universal hash function

### 3.2. Analyzing bottlenecks

Cryptographic algorithms use large bit-width operations to improve security. For example, UMAC uses 32-bit and 64-bit additions and 32-bit multiplications. RC6 uses 32-bit additions and multiplications [11]. Let us now analyze bottlenecks in speeding up wide operand data paths.

- The hardware complexity of an w-bit array multiplier or a w-bit parallel multiplier increases as (O(w^2)). The hardware-complexity of a w-bit adder increases as (O(w)) [3]. Furthermore, the area of an adder is much smaller than that of a multiplier for operand sizes larger than 8-bits, so the hardware-complexity of the NH data path increases as C_1w^2+C_2w+C_3. C_1 is equal to 0.5266; C_2 is equal to 4.6782 and C_3 is equal to -5.0784 when the NH universal hash data path is implemented.
on the targeted Virtex II device. The area of adder, multiplier and NH universal hash data path is shown in Figure 2(a). For example, while an 8-bit multiplier consumes 44 Virtex slices, a 16-bit multiplier consumes 161 Virtex slices and a 32-bit multiplier consumes 588 Virtex slices.

- The delay of a w-bit array multiplier increase as \(O(w)\) [3]. Since the clock rate of an adder is faster than that of a multiplier in a NH universal hash data path, multiplier delay determines the critical path of NH universal hash data path. The delay of the critical path in a NH universal hash data path is \(D_1w + D_2\) ns and the maximum clock rate is \(\frac{1}{D_1w + D_2}\) MHz. \(D_1\) is equal to 0.1296; \(D_2\) is equal to 2.1220 when the NH universal hash data path is implemented on the targeted Virtex II device. The clock rates of 4-bit, 8-bit, 16-bit and 32-bit multipliers when implemented on the Virtex II device are 353 MHz, 310 MHz, 237 MHz and 160 MHz respectively as shown in Figure 2(b).

- The area of w-bit multiplier when implemented on Xilinx Virtex II FPGA device is \(0.5139w^2 + 2.0629w - 4.1765\) slices and the maximum clock rate it can achieve is \(\frac{1}{0.1296w + 2.1220}\) MHz. The area of w-bit adder when implemented on targeted FPGA device is \(0.5w\) slices and the maximum clock rate it can achieve is \(\frac{1}{0.468w + 2.5268}\) MHz. By using these equations, Figure 2(c) shows that the normalized throughput/area ratio (using the ratio for a 2-bit unit as 1) for multipliers and the NH universal hash decreases quadratically with operand size \(w\). The case for adders is less dramatic. Since area of a NH universal hash data path is dominated by the area of the multiplier and the throughput of a NH universal hash data path is determined by that of the multiplier, the curves of multiplier and NH hash are almost same.
Figure 2: (a) Area and (b) Throughput and (c) Throughput/Area ratio of adders, multipliers and NH universal hash data path as a function of operand size
3.3. Reducing Collision Probability of NH Universal Hash [14] [15] [16] [17]

When implemented in software, the bit-width $w$ of the NH universal hash function is determined
by the architecture of the underlying processor. Increasing $w$ is not a feasible solution to reduce the
collision probability. However, since NH is a universal hash function, its collision probability of $2^{-w}$
can be reduced to $2^{-nw}$ by hashing the same message $n$ times using $n$ independent subkeys and
concatenating the results. If we hash a message twice using the $w$-bit NH universal hash function, each
time with a different set of subkeys and concatenating the two hash values, the collision probability
will drop from $2^{-w}$ to $2^{-2w}$. Furthermore, the resulting new hash function is as resistant as the old one to
cryptographic attacks.

However this solution requires twice subkey material. [14] describes Toeplitz extension to
reduce the amount of subkey material making this approach practical. As shown in Figure 3, when we
use two 16-bit data paths to construct a 32-bit NH universal hash using Toeplitz extension, the subkeys
for the second data path are obtained by shifting the corresponding subkeys of the first data path. When
Toeplitz extension is used a single 1024 $w$-bit-word subkey RAM is sufficient independent of the
number of data paths. The proof that the Toeplitz extension does not compromise the collision
probability property is also presented in [14].

3.4. Our Contribution

We propose to divide a $2w$-bit data path into two $w$-bit data paths and concatenate their results to
construct an equivalent $2w$-bit data path. The concept of equivalence is crucial. Obviously, a
straightforward data path and the corresponding divide-and-concatenate data path cannot be equivalent
in terms of the results that they output. We define two data paths to be equivalent if the results that they
output satisfy a pre-defined property. For universal hash functions and associated message
authentication codes the actual result is not important. Rather, it is the collision probability of the result
that is important. Hence, we propose that two data paths implementing a hash function be considered
equivalent if they have the same collision probability. When we discuss equivalent data paths and
architectures in section 4, it means 1) they can process same size input every clock cycle and 2) they
have the same collision probability.

4. Divide and Concatenate: An Architecture Level Optimization Technique

The straightforward 32-bit NH universal hash data path shown in Figure 1 takes two 32-bit
message words every cycle and generates a 64-bit hash value after the entire message is processed.

Using the divide-and-concatenate technique a 32-bit NH universal hash data path with a collision
probability of $2^{-32}$ can be constructed using two 16-bit NH universal hash data paths, each with collision
probability of $2^{-16}$, and concatenating their 32-bit results to generate a 64-bit hash value. This
corresponds to the two data paths at the top of Figure 3. However, it can only process 32-bit input every
cycle. The duplicated divide-and-concatenate architecture shown in Figure 3 uses four 16-bit NH
universal hash data paths and processes a 64-bit input every cycle (same as the straightforward 32-bit
NH data path). The fixed shift operation S and the concatenation operation $\| \|$ in Figure 3 just rename the
wires in the circuit and do not contribute to the area overhead in hardware. This divide-and-concatenate
architecture using four $\frac{w}{2}$-bit data paths is equivalent to a single w-bit straightforward data path in two
aspects: 1) they can process 2w-bit input in every clock cycle. 2) they have the collision probability of
As we discussed in section 3.2, the hardware complexity of a straightforward w-bit NH universal hash data path as shown in Figure 3 is $C_1w^2+C_2w+C_3$ slices. The maximum clock rate of straightforward NH universal hash data path is $\frac{1}{D_1w + D_2}$ MHz. The throughput is $\frac{2w}{D_1w + D_2}$ Mbps, because each clock it can process 2w-bit input. We have the throughput/area ratio for straightforward NH universal hash as follows:

$$\text{Throughput} / \text{Area} = \frac{2w}{(C_1w^2 + C_2w + C_3)(D_1w + D_2)} \quad (1)$$

The area of the divide-and-concatenate architecture that uses four w/2 NH universal hash is $4(C_1(w/2)^2+C_2(w/2)+C_3)$. The throughput is $\frac{2w}{D_1\frac{w}{2} + D_2}$ and the throughput/area ratio for this divide-and-concatenate architecture is:

$$\text{Throughput} / \text{Area} = \frac{4w}{(D_1w + 2D_2)(2C_1w^3 + 4C_2w + 8C_3)} \quad (2)$$

For the targeted Xilinx FPGA device, $C_1$, $C_2$, $C_3$, $D_1$ and $D_2$ is equal to 0.5266, 4.6782, -5.0784, 0.1296 and 2.1220 respectively and the throughput of the straightforward 32-bit NH universal hash data path is 10.24 Gbps. The throughput of the equivalent 16-bit divide-and-concatenate data path is 15.17 Gbps. The equivalent 16-bit divide-and-concatenate data path consumes 816 FPGA slices compared to 684 slices by the straightforward 32-bit data path. Compared to the straightforward 32-bit NH universal hash data path, the equivalent 16-bit divide-and-concatenate data path yields a 48% improvement in throughput with an associated area overhead of 19%.
Using equation (1) and (2), the throughput/area ratio of 0.0186 Gbps/slice for the divide-and-concatenate architecture is 25% more efficient than 0.0149 Gbps/slice for the straightforward architecture. The divide-and-concatenate architecture generates more throughput per slice.

Figure 3: NH data path composed of four 16-bit NH universal hash data paths with a collision probability of $2^{-32}$

Generalizing the area, throughput and throughput/area ratio of equivalent divide-and-concatenate architectures when 8-bit, 4-bit and 2-bit data paths are used to construct the equivalent divide-and-concatenate architecture, we have:

$$\text{Area} = n^2 \times (C_1(w/n)^2 + C_2(w/n) + C_3), \quad \text{Throughput} = \frac{2w}{D_1 \frac{w}{n} + D_2}, \quad n=1, 2, 4, 8, 16 \quad (3)$$
In equation (3) and (4), \( \frac{w}{n} \) defines the width of data path used in the divide-and-concatenate architecture. When \( n \) is equal to 1, one \( w \)-bit NH universal hash data path is used and it is the straightforward implementation and equation (4) can be simplified to equation (1). When \( n \) is equal to 2, four \( \frac{w}{2} \)-bit NH universal hash data paths are used to construct the equivalent divide-and-concatenate architecture and equation (4) can be simplified to equation (2).

Which equivalent divide-and-concatenate architecture is the most efficient one? We can find out the maximum value of this throughput/area ratio by differentiating equation (4) based on the independent variable \( n \) and setting it equal to zero. We have:

\[
2C_3D_2n^3 + (C_2D_2w + C_3D_1w)n^2 + (C_1D_2w^2 + C_2D_1w^2)n + C_1D_1w^3 = 0
\]  

Solving equation (5), we get 3.12 at which equation (4) reaches its maximum value. The divide-and-concatenate architecture using \( \frac{w}{n} \)-bit data paths is the most efficient architecture. Since \( \frac{32}{3.12} = 10.3 \) does not have physical meaning, 8-bit data path can be used to construct the most efficient architecture.

Table 1 summarizes the area, throughput and throughput/area ratio for five equivalent 32-bit NH universal hash data paths (i.e., all these data paths have a collision probability of \( 2^{-32} \) and process 64 input bits every clock cycle). The 8-bit data path has the best throughput/area ratio. It achieves 90% throughput improvement with only 40% area overhead.
Table 1: Maximum clock rate, input word size, area, throughput and throughput/area ratio of five equivalent divided-and-concatenated data paths for a NH universal hash with collision probability of $2^{-32}$.

<table>
<thead>
<tr>
<th>Size</th>
<th>2bit</th>
<th>4bit</th>
<th>8bit</th>
<th>16bit</th>
<th>32bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum clock rate(MHz)</td>
<td>420</td>
<td>353</td>
<td>310</td>
<td>237</td>
<td>160</td>
</tr>
<tr>
<td>Input word size</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Throughput (Gbps)</td>
<td>26.9</td>
<td>22.6</td>
<td>19.8</td>
<td>15.2</td>
<td>10.2</td>
</tr>
<tr>
<td>Area (Slice)</td>
<td>1634</td>
<td>1408</td>
<td>1056</td>
<td>818</td>
<td>684</td>
</tr>
<tr>
<td>Throughput/area(Gbps/slice)</td>
<td>0.0164</td>
<td>0.0159</td>
<td>0.0188</td>
<td>0.0186</td>
<td>0.0149</td>
</tr>
</tbody>
</table>

When constructing equivalent divide-and-concatenate architecture using 4-bit NH universal hash data path, sixty four 4-bit data paths are required, but the key word length is only 4 and we can not shift it 8 times using Toeplitz-extension and we have to use 8 times key material. In equivalent divide-and-concatenate architecture using 2-bit NH universal hash data path, 16 times key material are required.

Equation (5) can be used to find out the most efficient divide-and-concatenate architecture for any ASIC and FPGA library using appropriate parameter when a higher collision probability is required. When $w$ is equal to 64 and 128, the divide-and-concatenate architecture using 8-bit data paths is still the most efficient architecture.

4-bit and 2-bit data path based designs are not as efficient as an 8-bit data path based design. In the divide-and-concatenate approach, the number of adders increases quadratically while their area
decreases linearly as the word size increases. In 8-bit and larger designs the NH universal hash area is dominated by large multipliers. On the other hand, in 4-bit and 2-bit data path based designs the NH area is dominated by adders. This is because the area of an adder is comparable to that of a multiplier and the number of adders grows quadratically.

A drawback of the divide-and-concatenate technique is that the length of the output hash doubles as you go from the straightforward 32-bit data path to an equivalent 16-bit data path to an equivalent 8-bit data path and so on. In fact, the length of hash value of an equivalent 2-bit data path is 16 times longer than that for the straightforward data path. We do not apply this technique to 1-bit NH universal hash, which requires much key material and generates output that is too wide. We will discuss how this increase in output length impacts the overall performance of UMAC in next section.

5. The UMAC Architecture

The Universal Message Authentication Code has three steps [14]:

• Step 1: subkey generation:

In this step, the user key is expanded into (i) 1024 w-bit-word subkeys that are used by NH universal hash in step 2 and (ii) 512-bit key A for the HMAC-SHA1 cryptographic primitive used in step 3. Since subkeys are generated just once at the beginning of a session, this step does not impact the overall throughput of UMAC.

• Step 2: Hashing the input message using NH universal hash:

If the message is larger than 1024 w-bit words, it will be partitioned into 1024 w-bit word blocks. Each block is compressed independently according to the method explained in section 2.3 and the final
hash value is obtained as the concatenation of the hash values of each block and the message length encoded in binary:

\[ HM = NH(\text{Message block 1})||NH(\text{Message block 1})\ldots||NH(\text{Message block 1 t})||\text{Message Length} \]

- **Step 3: Computing the MAC:**

  In universal hash based MACs, a cryptographic primitive is needed to generate a MAC from compressed hash value. In UMAC, HMAC-SHA1 is applied to the concatenated hash HM from step 2 to obtain fixed length MAC.

  \[ \text{MAC} = \text{HMAC-SHA1}_A(HM || \text{Nonce}) \]

  Typically the nonce is a counter which the sender increments with each transmitted message. \( A \) is the key of HMAC-SHA1 and was generated in step 1.

  HMAC-SHA1 operates on 512-bit block and generates a 160-bit result after 80 cycles. This 160-bit result is used by HMAC-SHA1 when it processes the next 512-bit message block. When all message blocks are consumed, this 160-bit result is the MAC. An implementation of HMAC-SHA1 on an FPGA yields a throughput of 652 Mbps and consumes 569 slices [12].

  When the UMAC architecture processes a single message stream, multiple NH data paths hash different parts of the message and concatenate them into HM as described in step 2. HM is then hashed into a MAC using HMAC-SHA1. The divide-and-concatenate approach cannot be applied to HMAC-SHA1 as it is not a universal hash function.

  In general, the throughput of HMAC-SHA1 is not a bottleneck for the overall UMAC architecture. The straightforward 32-bit NH universal hash data path compresses each \( 1024 \times 32 \)-bit message block
input into a 64-bit intermediate value or hash value output yielding a compression ratio of 512. The throughput of this straightforward NH universal hash data path is 10.24 Gbps at its input and 20 Mbps (=10.24 Gbps÷512) at its output (which is also the input to HMAC-SHA1). This is 32 times smaller than the 652 Mbps throughput of HMAC-SHA1. The 2-bit equivalent NH universal hash data path generates an output of 1024 bits translating into a compression ratio of only 32 (=1024×32-bit input ÷1024-bit output). The resulting throughput of 840 Mbps (=26.9 Gbps÷32) at the input to the HMAC-SHA1 is greater than 652 Mbps throughput of HMAC-SHA1. Now, HMAC-SHA1 is the bottleneck. Table 2 summarizes the throughput, compression ratio and output data rate of the equivalent data paths for the 32-bit NH universal hash.

Table 2: Input throughput, compression ratio, output data rate for equivalent architectures of a 32-bit NH universal hash.

<table>
<thead>
<tr>
<th>Size</th>
<th>2bit</th>
<th>4bit</th>
<th>8bit</th>
<th>16bit</th>
<th>32bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput (Gbps)</td>
<td>26.9</td>
<td>22.6</td>
<td>19.8</td>
<td>15.2</td>
<td>10.2</td>
</tr>
<tr>
<td>Compression ratio</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>Output data rate (Mbps)</td>
<td>840</td>
<td>353</td>
<td>155</td>
<td>60</td>
<td>20</td>
</tr>
</tbody>
</table>

To effectively utilize HMAC-SHA1 data path, the UMAC architecture can be designed such that HMAC-SHA1 works with multiple message streams and associated NH universal hash data paths as shown in Figure 4. The input data is separated into multiple streams by demultiplexer (DEMUX) and each stream is sent to an equivalent divide-and-concatenate data path. The hash value of each NH data path is then sent to HMAC-SHA-1 by multiplexer (MUX). For example, since the data rate at the
output of straightforward 32-bit NH universal hash architecture is 20 Mbps and the throughput of HMAC-SHA1 is 652 Mbps, one HMAC-SHA1 data path can work with thirty two 32-bit NH universal hash data paths and associated message streams. This translates into an effective UMAC throughput of 326.4 Gbps.

Table 3: Effective throughput, area and throughput/area ratio of four UMAC architectures.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>4-bit</th>
<th>8-bit</th>
<th>16-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max throughput (Gbps)</td>
<td>22.6</td>
<td>79.2</td>
<td>167.2</td>
<td>326.4</td>
</tr>
<tr>
<td>Area(slices)</td>
<td>1849</td>
<td>3840</td>
<td>9545</td>
<td>22457</td>
</tr>
<tr>
<td>Throughput/Area (Gbps/slice)</td>
<td>0.012</td>
<td>0.021</td>
<td>0.018</td>
<td>0.015</td>
</tr>
</tbody>
</table>

Table 3 summarizes the maximum throughput, area (NH universal hash data path + HMAC-SHA1 data path) and throughput/area ratio of four UMAC architectures. For example, for 8-bit equivalent
UMAC architecture, we use 4 (≈652 Mbps ÷ 155 Mbps) 16-bit NH universal hash data paths with 960 slices. Its area is 3840 slices (=960 slices per equivalent NH universal hash × 4+569 slices for HMAC-SHA1). Its throughput is 79.2 Gbps (19.8 Gbps per equivalent NH universal hash data path× 4). The 8-bit equivalent architecture has the best throughput/area ratio.

6. Conclusions

Applying general hardware design techniques to cryptographic architectures yields only moderate improvements. We defined a collision probability equivalent data path and combined it with the divide-and-concatenate technique to design high throughput architectures for UMAC calculations based on using multiple small multipliers in place of one larger multiplier. We characterized the area and throughput of equivalent data paths of 32-bit NH universal hash and demonstrated that the 8-bit equivalent NH universal hash data path is the most efficient architecture with the associated UMAC architecture achieving 79.2 Gbps using only 3840 FPGA slices. The area of a multiplier is determined by the algorithm and it increases quadratically as the operand size is increased without depending on the manufacturing technology. This divide-and-concatenate technique can be applied to any other FPGA device or ASIC implementation. When a w-bit parallel multiplier is used in NH universal hash data path whose delay is increases as O(log(w)), we can use the same method discussed in section 4 to find out the most efficient divide-and-concatenate architecture.

The divide-and-concatenate technique can not speed-up software implementations but can only improve the collision probability beyond that provided by the processor architecture. This is because, if a processor supports w-bit additions and multiplications in one or two cycles, then w/2-bit operations
will also consume the same number of cycles as w-bit operations. This divide-and-concatenate technique also does not work on hash functions SHA-1 and MD5, because they are not universal hash functions and their collision probabilities do not have the same feature of universal hash functions discussed in this paper.

References


