A Fast Hash Family for Memory Integrity

Qiming Li, Sampo Sovio

***Abstract*—We give a first construction of an ϵ-balanced hash family based on linear transformations of vectors in $\mathbb{F}_2$, where $\epsilon = 1/(2^n - 1)$ for n-bit hash values, regardless of the message size. The parameter n is also the bit length of the input blocks and the internal state, and can be chosen arbitrarily without design changes, This hash family is fast, easily parallelized, and requires no initial setup. A secure message authentication code can be obtained by combining the hash family with a pseudo random function. These features make the hash family attractive for memory integrity protection, while allowing generic use cases.**

***Index Terms*—ϵ-balanced hash family, message authentication code, memory integrity**

I. Introduction

HASH functions and Message Authentication Codes (MAC) play important roles in information integrity protection. A MAC is an algorithm that computes an authentication tag from an input message using a secret key, and the tag is then stored or transmitted together with the message. The integrity of the message can later be verified by computing a tag again using the same key and checking if it matches the previous tag that has been stored or transmitted. A MAC is usually said to be secure if it is computationally infeasible for an adversary, with varying assumptions on computing capabilities, such a limited number of accesses to an oracle, to compute an unseen message and a tag that allow successful verification without knowing the key in advance.

Krawczyk [5] showed that a MAC can be constructed by first applying a traditional hash function on the input messages and encrypting the result. The resulting MAC is secure if the encryption scheme is secure and the hash function is randomly chosen from an ϵ-balanced hash family, where the probability of a randomly chosen hash function maps any given message to a hash value is bounded by ϵ. Other schemes such as UMAC [3] may require that the hash family is ϵ-universal, where the probability that a randomly chosen hash function gives the same hash value for two distinct messages is no more than ϵ. The notion of ϵ-universal hash families is generalized from *Universal*$_2$ hash families (Carter and Wegman [4]), where a hash family is called Universal$_2$ if it is 2^{-n}-universal for n-bit hash values.

The hash families used in [5], [6] are based on linear transformations on the input message, including division by a polynomial or multiplication by a Toeplitz matrix. As observed by Shoup [9], the bound ϵ under such schemes increases linearly with the number of blocks in the input message. In other words, the advantages of an adversary in breaking the MAC increases proportionally as the message size grows. A similar effect can be observed for other polynomial division based schemes such as GCM encryption [7].

A theoretical nested MAC (NMAC) construction was later proposed by Bellare, Canetti and Krawczyk [2], where a tag is computed by first applying a hash family then a pseudo random function (PRF). The popular HMAC scheme is a concrete realization where both the inner hash family and the outer PRF are based on cryptographic one-way hash functions such as SHA-1. It was shown [2], [1] that the NMAC/HMAC scheme is secure if the PRF is secure and the inner hash family is computational ϵ-universal. The advantage of an adversary is similarly amplified proportionally with the message size, as in the case with hash families with information-theoretic bounds.

It is worth to note that such proportional amplification of the adversarial advantage can be avoided by using a Universal$_2$ hash family. However, known Universal$_2$ hash families require a hash key whose length is proportional to the message length.

On the other hand, we observe that protection of the integrity of the content stored in computer memory, such as CPU instructions and dynamic data, poses different challenges compared with generic use cases such as file integrity protection. First, the performance of such schemes is a major concern. As a result, it is desirable to have a block size that matches the native memory word size, such as 32 bits or 64 bits in modern CPU architectures, to avoid buffering and padding. Furthermore, while the requirement on the hash collision probability can be somewhat relaxed in practical memory protection use cases (say, 2^{-30} is considered sufficient), it is desirable to have a bound that is independent of the message size or the number of messages to be hashed using a single key. In addition, such a hash family should allow efficient hardware implementations and easy integration into existing memory and CPU architectures, where parallelization and pipelines may be helpful. Last but not the least, it is desirable to configure the hash family such that the security level evolves naturally as the computer architecture advances, instead of requiring redesigns for newer CPUs or larger memory sizes.

Therefore, it is interesting to investigate if it is possible to construct an ϵ-balanced hash family for an ϵ as close to the optimal 2^{-n} as possible, where the key size and the ϵ bound does not depend on the input size, since such a hash family would imply a practical and secure MAC, and perhaps many other hash-based security schemes, where the adversarial advantage does not increase proportionally with the input size.

Contributions*:* In this paper we propose an ϵ-balanced (and ϵ-universal) hash family for $\epsilon = 1/(2^n - 1)$ with the following characteristics.

1) Flexible block size n to match the target computer architecture.
2) The parameter n also defines the key size (n^2 bits), the bound ϵ, and the internal state size (n bits), regardless

Authors are with Finland R&D Center, Huawei Technologies Oy (Finland) Co. Ltd., Helsinki, Finland.

of message size.

3) Fast software/hardware implementations with only logical x-or operations.
4) Easily parallelized and integrated to pipelines.

These features makes the proposed hash family attractive for constructing MAC schemes using known composition methods such as NMAC and UMAC for memory integrity protection, while allowing generic use cases.

II. Notations

Capital letters in bold font (such as $\mathbf{K}$) denote matrices, and small letters in bold font (such as $\mathbf{m}$) denote vectors. Small letters in italic font (such as n) denote scalars. Matrices enclosed by square brackets, such as $[\mathbf{M}_1\ \mathbf{M}_2]$, refer to an augmented matrix formed by horizontally joining the columns of $\mathbf{M}_1$ and $\mathbf{M}_2$, assuming the number of rows in $\mathbf{M}_1$ and $\mathbf{M}_2$ are the same. Matrices (or vectors) enclosed by parenthesis, such as $(\mathbf{v}_1, \mathbf{v}_2)$, refer to an augmented matrix or vector formed by vertically joining the rows of $\mathbf{v}_1$ and $\mathbf{v}_2$. We use vertical bars, such as $|\mathbf{v}|$, to denote the length of the vector $\mathbf{v}$ (i.e., the number of coordinates), and vertical bars with a subscript $|\mathbf{v}|_n$ to mean the length of the vector in number of blocks of n coordinates.

We denote by $\mathbb{B}$ the set of Boolean values, i.e., $\mathbb{B} = \{0, 1\}$, and by $\mathbb{B}^n$ the vector space of Boolean vectors of n coordinates, where n is referred to as the block size. A data block of n bits is interpreted as a vector in $\mathbb{B}^n$. Similarly, we denote by $\mathbb{B}^{rn}$ the vector space representation of r data blocks, and $\mathbb{B}^{n*}$ the set of vectors in all vector spaces $\mathbb{B}^{rn}$ for integer $r \geq 1$, which forms the message space of the proposed hash family.

We denote by $\mathcal{N}$ the set of n-by-n nonsingular (invertible) Boolean matrices, which is the key space for our schemes.

III. A Hash Family

A. A Basic Construction

For $n \geq 1$ and a constant nonzero initial state vector $\mathbf{s}_0 \in \mathbb{B}^n$, we define the hash family $H : \mathcal{N} \times \mathbb{B}^{n*} \to \mathbb{B}^n$ such that, for any message $\mathbf{m} \in \mathbb{B}^{n*}$, let $r = |\mathbf{m}|_n$ be the number of blocks in $\mathbf{m}$, the hash value of $\mathbf{m}$ is given by

$$H(\mathbf{K}, \mathbf{m}) = [\mathbf{K}^r\ \mathbf{K}^{r-1}\ \ldots\ \mathbf{K}]\mathbf{m} + \mathbf{K}^r\mathbf{s}_0$$

where the multiplications and additions are done over $\mathbb{F}_2$. We can write $\mathbf{m} = (\mathbf{m}_1, \mathbf{m}_2, \ldots, \mathbf{m}_r)$ as a representation in n-bit blocks. Expanding the left multiplication of the augmented key matrix $[\mathbf{K}^r\ \mathbf{K}^{r-1}\ \ldots\ \mathbf{K}]$ with the message $\mathbf{m}$, and reorganizing the terms, the hash family is equivalently

$$\begin{aligned} H(\mathbf{K}, \mathbf{m}) &= [\mathbf{K}^r\ \mathbf{K}^{r-1}\ \ldots\ \mathbf{K}]\mathbf{m} + \mathbf{K}^r\mathbf{s}_0 \\ &= \mathbf{K}^r\mathbf{m}_1 + \mathbf{K}^{r-1}\mathbf{m}_2 + \cdots + \mathbf{K}\mathbf{m}_r + \mathbf{K}^r\mathbf{s}_0 \\ &= [\mathbf{K}^r\ \mathbf{K}^{r-1}\ \ldots\ \mathbf{K}](\mathbf{m}_1 + \mathbf{s}_0, \mathbf{m}_2, \ldots, \mathbf{m}_r) \end{aligned}$$

Furthermore, let $\mathbf{m}'_i = (\mathbf{m}_1, \mathbf{m}_2, \ldots, \mathbf{m}_i)$ be the sub-vector composed from $\mathbf{m}_1$ to $\mathbf{m}_i$ for $1 \leq i \leq r$. We note that the hash family can also be defined recursively as $H(\mathbf{K}, \mathbf{m}'_i) = \mathbf{K}(H(\mathbf{K}, \mathbf{m}'_{i-1}) + \mathbf{m}_i)$ for $i \geq 1$, and $H(\mathbf{K}, \mathbf{m}_0) = \mathbf{s}_0$.

In other words, to compute the hash, we start from a state vector $\mathbf{s}$ that is initially $\mathbf{s}_0$, and for each message block $\mathbf{m}_i$, we add $\mathbf{m}_i$ to $\mathbf{s}$ and left multiply the result by $\mathbf{K}$ to yield the next state. The hash value is simply the final state vector after all message blocks are processed. This gives the pseudo-code in Algorithm 1 for computing H.

Algorithm 1 Basic Hash

```
1: procedure HASH($\mathbf{s}_0$, $\mathbf{K}$, $\mathbf{m}$)
2:     $\mathbf{s} \leftarrow \mathbf{s}_0$
3:     for $i = 1, 2, \ldots, r$ do
4:         $\mathbf{s} \leftarrow \mathbf{K}(\mathbf{s} + \mathbf{m}_i)$
5:     end for
6:     return $\mathbf{s}$
7: end procedure
```

Since the arithmetic is in $\mathbb{F}_2$, the addition in line 4 is simply an x-or between the two vectors, and the matrix multiplication is similarly done by at most $n - 1$ x-or of the n-bit column vectors of $\mathbf{K}$.

a) Main Result: The hash family H as defined above is ϵ-balanced and ϵ-universal for $\epsilon = 1/(2^n - 1)$, regardless of the message size. Detailed analysis is given in Section IV.

B. Parallelization

An advantage of the proposed hash family is that the computation of messages with multiple blocks can be easily parallelized.

Recall that $|\mathbf{v}|_n$ refers to the number of n-bit blocks in vector $\mathbf{v}$. For an r-block messages $\mathbf{m}$ (i.e., $|\mathbf{m}|_n = r$), we can choose a parameter $p \leq r$ and break $\mathbf{m}$ into p chunks of at most t blocks each. We write $\mathbf{m} = (\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_p)$, where $|\mathbf{c}_i|_n = t$ for $2 \leq i \leq p$ and $|\mathbf{c}_1|_n = r - (p-1)t$. In other words, all chunks except the first have the same size.

Then, we can process the chunks in parallel, and combine the results to get the final hash. For example, to compute the hash value of $\mathbf{m}$ given initial state $\mathbf{s}_0$ and randomly chosen key $\mathbf{K}$, a bottom up approach (Algorithm 2) groups every t blocks into a chunk, computes the hash value of all the chunks in parallel, and compute the hash value of the resulting data (line 11) again using the same bottom up approach, until the number of chunks is 1.

Algorithm 2 Parallel Hash (Bottom-Up)

```
1: procedure PHASH1($\mathbf{s}_0$, $t$, $\mathbf{K}$, $\mathbf{m}$)
2:     $p \leftarrow \lceil |\mathbf{m}|_n / t \rceil$    ▷ $p$ chunks of at most $t$ blocks each
3:     if $p = 1$ then
4:         return HASH($\mathbf{s}_0, \mathbf{K}, \mathbf{m}$)
5:     end if
6:     $(\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_p) \leftarrow \mathbf{m}$, where $|\mathbf{c}_i|_n = t$ for $2 \leq i \leq p$.
7:     $\mathbf{d}_1 \leftarrow$ HASH($\mathbf{s}_0, \mathbf{K}, \mathbf{c}_1$)
8:     for $i = 2, 3, \ldots, p$ do                 ⎫
9:         $\mathbf{d}_i \leftarrow$ HASH($\mathbf{0}, \mathbf{K}, \mathbf{c}_i$)          ⎬ in parallel
10:    end for                                      ⎭
11:    return PHASH1($\mathbf{0}, t, \mathbf{K}^t, (\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_{p-1})$) + $\mathbf{d}_p$
12: end procedure
```

The correctness of Algorithm 2 follows from the identity below.

$$
\begin{aligned}
& \textsc{Hash}(\mathbf{s}_0, \mathbf{K}, m) \\
= \; & [\mathbf{K}^r \; \mathbf{K}^{r-1} \; \cdots \; \mathbf{K}](\mathbf{m}_1 + \mathbf{s}_0, \mathbf{m}_2, \cdots, \mathbf{m}_r) \\
= \; & [\mathbf{K}^r \; \cdots \; \mathbf{K}^{(p-1)t+1}](\mathbf{m}_1 + \mathbf{s}_0, \cdots, \mathbf{m}_{r-(p-1)t}) + \\
& [\mathbf{K}^{(p-1)t} \; \cdots \; \mathbf{K}^{(p-2)t+1}](\mathbf{m}_{r-(p-1)t+1}, \cdots, \mathbf{m}_{r-(p-2)t}) \\
& + \cdots + \\
& [\mathbf{K}^{2t} \; \cdots \; \mathbf{K}^{t+1}](\mathbf{m}_{r-2t+1}, \cdots, \mathbf{m}_{r-t}) + \\
& [\mathbf{K}^{t} \; \cdots \; \mathbf{K}](\mathbf{m}_{r-t+1}, \cdots, \mathbf{m}_{r}) \\
= \; & \mathbf{K}^{(p-1)t}[\mathbf{K}^{r-(p-1)t} \; \cdots \; K]\mathbf{c}_1 \; + \mathbf{K}^{(p-2)t}[\mathbf{K}^{t} \; \cdots \; K]\mathbf{c}_2 \\
& + \cdots + \\
& \mathbf{K}^{t}[\mathbf{K}^{t} \; \cdots \; K]\mathbf{c}_{p-1} \; + \; [\mathbf{K}^{t} \; \cdots \; K]\mathbf{c}_p \\
= \; & \mathbf{K}^{(p-1)t}\mathbf{d}_1 \; + \mathbf{K}^{(p-2)t}\mathbf{d}_2 \; + \cdots + \mathbf{K}^{t}\mathbf{d}_{p-1} \; + \; \mathbf{d}_p \\
= \; & [\mathbf{K}^{(p-1)t} \; \cdots \; \mathbf{K}^{t}](\mathbf{d}_1, \mathbf{d}_2, \cdots, \mathbf{d}_{p-1}) + \mathbf{d}_p \\
= \; & \textsc{Hash}(\mathbf{0}, \mathbf{K}^t, (\mathbf{d}_1, \mathbf{d}_2, \cdots, \mathbf{d}_{p-1})) + \mathbf{d}_p
\end{aligned}
$$

On the other hand, we can also follow a top-down approach and repeatedly break a long message into p chunks until the message is no more than p_0 blocks for some threshold $p_0 \geq p$. This gives the top-down version of the parallel hashing (Algorithm 3).

Algorithm 3 Parallel Hash (Top-Down)

```
1:  procedure PHASH2($\mathbf{s}_0$, $p_0$, $p$, $\mathbf{K}$, $\mathbf{m}$)
2:      if $|\mathbf{m}|_n \leq p_0$ then
3:          return HASH($\mathbf{s}_0, \mathbf{K}, \mathbf{m}$)
4:      end if
5:      $t \leftarrow \lceil |\mathbf{m}|_n / p \rceil$    ▷ $p$ chunks of at most $t$ blocks each
6:      $(\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_p) \leftarrow \mathbf{m}$, where $|\mathbf{c}_i|_n = t$ for $2 \leq i \leq p$.
7:      $\mathbf{d}_1 \leftarrow$ PHASH2($\mathbf{s}_0, p_0, p, \mathbf{K}, \mathbf{c}_1$)
8:      for $i = 2, 3, \ldots, p$ do                        }
9:          $\mathbf{d}_i \leftarrow$ PHASH2($\mathbf{0}, p_0, p, \mathbf{K}, \mathbf{c}_i$)   } in parallel
10:     end for                                             }
11:     return PHASH2($\mathbf{0}, p_0, p, \mathbf{K}^t, (\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_{p-1})$) + $\mathbf{d}_p$
12: end procedure
```

Note that there can be many variations of the above mentioned algorithms. For example, instead of using fixed parameters such as p, t and p_0, such parameters can be dynamically chosen according to the circumstances. Furthermore, in some cases it may be beneficial to apply a hybrid approach, where intermediate hash values can be computed from bottom-up while the entire data message is broken down and processed in a top-down manner.

IV. Analysis

As shown in [2], [3], a secure MAC can be constructed from an ϵ-balanced (ϵ-universal) hash family and a pseudo random function (PRF). To analyze the security of such a secure MAC using the hash family H as the hash family, it suffices to show that H is ϵ-balanced and ϵ-universal.

More precisely, for some message space $\mathcal{M}$ and key space $\mathcal{K}$, we have

Definition 1. *A hash family $H : \mathcal{K} \times \mathcal{M} \to \mathbb{B}^n$ is ϵ-balanced if for any $\mathbf{x} \in \mathcal{M}$ and $\mathbf{y} \in \mathbb{B}^n$, $\Pr[H(\mathbf{K}, \mathbf{x}) = \mathbf{y}] \leq \epsilon$, where $\mathbf{K}$ is uniformly chosen from $\mathcal{K}$ at random. Similarly, H is ϵ-universal if for any distinct $\mathbf{x}, \mathbf{w} \in \mathcal{M}$, $\Pr[H(\mathbf{K}, \mathbf{x}) = H(\mathbf{K}, \mathbf{w})] \leq \epsilon$,*

Essentially we follow the definitions in [4], [5], [6] with unsubstantial adaptations. In particular, we note that if a hash family is ϵ-balanced by Definition 1, then it is also ϵ-balanced by their definitions. Also note that two messages $\mathbf{x}$ and $\mathbf{w}$ are considered distinct if either their lengths differ, or $\mathbf{x} - \mathbf{w} \neq \mathbf{0}$. Furthermore, in some definitions found in prevoius work, the input of the hash family is limited to nonzero values. Such an assumption is useful in analysis and zero values are avoided by having nonzero initial values or paddings in practice.

As a preparatory step, we observe that, the inner product of any nonzero block with a uniformly random block is uniformly random.

Proposition 1. *For $n > 0$, any nonzero $\mathbf{x} \in \mathbb{B}^n$ and $b \in \mathbb{B}$, the number of vectors $\mathbf{k} \in \mathbb{B}^n$ such that $\mathbf{k} \cdot \mathbf{x} = b$ is 2^{n-1}.*

Proof. Let $\mathbf{k} = (k_1, \ldots, k_n)$, $\mathbf{x} = (x_1, \ldots, x_n)$ and let i be an index such that $x_i = 1$ $(1 \leq i \leq n)$. Now the condition $\mathbf{k} \cdot \mathbf{x} = b$ is equivalent to $k_i = b + \sum_{j=1, j \neq i}^{n} k_j x_j$. There are 2^{n-1} choices for the values of k_j, where $1 \leq j \leq n$ and $j \neq i$. For each choice, k_j is completely determined. Hence, the total number of choices for $\mathbf{k}$ is 2^{n-1} as claimed. □

Next we consider the likelihood that a matrix $\mathbf{K} \in \mathcal{N}$ happens to map one vector in $\mathbb{B}^n$ to another, where the vectors are not both zero.

Lemma 1. *For $n > 0$ and all $\mathbf{x}, \mathbf{y} \in \mathbb{B}^n$, where $\mathbf{x}$ and $\mathbf{y}$ are not both zero, the number of $\mathbf{K} \in \mathcal{N}$ such that $\mathbf{K}\mathbf{x} = \mathbf{y}$ is at most $N/(2^n - 1)$, where $N = \#\mathcal{N}$ is the total number of nonsingular Boolean matrices. That is, $\Pr[\mathbf{K}\mathbf{x} = \mathbf{y}] \leq 1/(2^n - 1)$ when $\mathbf{K}$ is uniformly chosen from $\mathcal{N}$ at random.*

Proof. First, it is easy to see that if one of $\mathbf{x}$ and $\mathbf{y}$ is zero, the other must be nonzero and there is no $\mathbf{K} \in \mathcal{N}$ that satisfies $\mathbf{K}\mathbf{x} = \mathbf{y}$. Hence, we only need to consider the cases where both $\mathbf{x}$ and $\mathbf{y}$ are nonzero.

We write $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ and $\mathbf{y} = (y_1, y_2, \ldots, y_n)$, where $x_i, y_i \in \mathbb{B}$. Without loss of generality, assume that $y_1 = 1$, since otherwise we can simply permute the rows of $\mathbf{K}$ and $\mathbf{y}$ simultaneously such that $y_1 = 1$, without affecting the number of possible candidates for $\mathbf{K}$.

From Proposition 1, the number of vectors $\mathbf{k} \in \mathbb{B}^n$ such that $\mathbf{k} \cdot \mathbf{x} = y_1$ is 2^{n-1}. Since $y_1 = 1$, such vectors do not include the zero vector and therefore are all possible candidates for the first row $\mathbf{k}_1$ of $\mathbf{K}$.

Now consider $\mathbf{k}_j$ the j-th row of $\mathbf{K}$, where $2 \leq j \leq n$. Similar to the first row, the number of vectors $\mathbf{k}$ such that $\mathbf{k} \cdot \mathbf{x} = y_j$ is 2^{n-1}. However, for $\mathbf{K}$ to be nonsingular, it is required that $\mathbf{k}_j$ is not a linear combination of all the previous rows $\mathbf{k}_1, \ldots, \mathbf{k}_{j-1}$.

Let $\mathbf{c} = (c_1, c_2, \ldots, c_{j-1})$ be the coefficient vector of a linear combination of the first $j-1$ rows, and clearly there are 2^{j-1} such linear combinations. Let $\widetilde{\mathbf{c}} = c_1\mathbf{k}_1 + \cdots + c_{j-1}\mathbf{k}_{j-1}$ be the result of such a linear combination.

We note that exactly half of the choices of $\mathbf{c}$ result in $\widetilde{\mathbf{c}}$ that satisfies $\widetilde{\mathbf{c}} \cdot \mathbf{x} = y_j$. To see this, we observe that for any such

$\widetilde{\mathbf{c}}$, we have

$$\widetilde{\mathbf{c}} \cdot \mathbf{x} = c_1\mathbf{k}_1\mathbf{x} + \cdots + c_{j-1}\mathbf{k}_{j-1}\mathbf{x} = c_1 y_1 + \cdots + c_{j-1}y_{j-1}$$

Since $y_1 = 1$, from Proposition 1 we know that the number of choices of $\mathbf{c}$ such that $\widetilde{\mathbf{c}} \cdot \mathbf{x} = y_j$ is 2^{j-2}.

Therefore, the number of candidate vectors for $\mathbf{k}_j$ such that $\mathbf{k}_j$ is not a linear combination of $\mathbf{k}_1, \ldots, \mathbf{k}_{j-1}$ and that $\mathbf{k}_j\mathbf{x} = y_j$ is $2^{n-1} - 2^{j-2}$.

Let N_k be the total number of candidates for $\mathbf{K}$ such that $\mathbf{Kx} = \mathbf{y}$, we have $N_k = 2^{n-1}(2^{n-1}-1)(2^{n-1}-2)\cdots(2^{n-1}-2^{n-2})$. Considering that the total number of n-by-n nonsingular Boolean matrices is $N = (2^n-1)(2^n-2)\cdots(2^n-2^{n-1})$, we have $N_k = N/(2^n-1)$ as claimed. □

Now we consider hash families $H_r : \mathcal{N} \times \mathbb{B}^{rn} \to \mathbb{B}^n$ defined below where the input messages are of the same lengths (r blocks). We observe that H_r is *linear* in the sense that for any two messages $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{B}^{rn}$, it holds for any key $\mathbf{K} \in \mathcal{N}$ and scalar c that $H_r(\mathbf{K}, \mathbf{x}_1) + H_r(\mathbf{K}, \mathbf{x}_2) = H_r(\mathbf{K}, (\mathbf{x}_1 + \mathbf{x}_2))$, and $H_r(\mathbf{K}, c\mathbf{x}) = cH_r(\mathbf{K}, \mathbf{x})$.

Lemma 2. *For $n, r > 0$, and $\epsilon = 1/(2^n-1)$, the hash family $H_r : \mathcal{N} \times \mathbb{B}^{rn} \to \mathbb{B}^n$ where $H_r(\mathbf{K}, \mathbf{m}) = [\mathbf{K}^r\ \mathbf{K}^{r-1}\ \cdots\ \mathbf{K}]\mathbf{m}$ is ϵ-balanced and ϵ-universal.*

Proof. Let $\mathbf{x}, \mathbf{w} \in \mathbb{B}^{rn}$ be distinct messages such that $\mathbf{x} \neq \mathbf{0}$, and $\mathbf{y} \in \mathbb{B}^n$ be any hash value. We write $\mathbf{x} = (\mathbf{x}'_{r-1}, \mathbf{x}_r)$, where $\mathbf{x}_r$ is the last n-bit block of $\mathbf{x}$, and $\mathbf{x}'_{r-1}$ is the prefix of $\mathbf{x}$ that contains all but the last block, We similarly write $\mathbf{w} = (\mathbf{w}'_{r-1}, \mathbf{w}_r)$.

For $r = 1$, it follows from Lemma 1 directly that $\Pr[H_1(\mathbf{K}, \mathbf{x}) = \mathbf{y}] = \Pr[\mathbf{Kx} = \mathbf{y}] \leq 1/(2^n-1)$. Hence, H_1 is ϵ-balanced. Also, H_1 is trivially ϵ-universal, since no $\mathbf{K}$ satisfies $\mathbf{Kx} = \mathbf{Kw}$ for distinct $\mathbf{x}$ and $\mathbf{w}$.

Now, assuming that H_{r-1} is ϵ-balanced and ϵ-universal for $r \geq 2$, we show that the bounds also apply to H_r. Since H_r is linear, we have

$$\begin{array}{cl} & H_r(\mathbf{K}, \mathbf{x}) = H_r(\mathbf{K}, \mathbf{w}) \\ \iff & \mathbf{K}(H_{r-1}(\mathbf{K}, \mathbf{x}'_{r-1}) + \mathbf{x}_r) \\ & = \mathbf{K}(H_{r-1}(\mathbf{K}, \mathbf{w}'_{r-1}) + \mathbf{w}_r) \\ \iff & H_{r-1}(\mathbf{K}, \mathbf{x}'_{r-1}) + \mathbf{x}_r = H_{r-1}(\mathbf{K}, \mathbf{w}'_{r-1}) + \mathbf{w}_r \\ \iff & H_{r-1}(\mathbf{K}, (\mathbf{x}'_{r-1} - \mathbf{w}'_{r-1})) = \mathbf{w}_r - \mathbf{x}_r \end{array}$$

Now, $\mathbf{x}'_{r-1} - \mathbf{w}'_{r-1} \neq \mathbf{0}$, since otherwise $\mathbf{x}_r - \mathbf{w}_r = H_{r-1}(\mathbf{K}, (\mathbf{x}'_{r-1} - \mathbf{w}'_{r-1})) = \mathbf{0}$, implying that $\mathbf{x} = \mathbf{w}$, contradicting with the assumption that $\mathbf{x}$ and $\mathbf{w}$ are distinct. By the induction hypothesis, we have

$$\begin{array}{cl} & \Pr[H_r(\mathbf{K}, \mathbf{x}) = H_r(\mathbf{K}, \mathbf{w})] \\ = & \Pr[H_{r-1}(\mathbf{K}, (\mathbf{x}'_{r-1} - \mathbf{w}'_{r-1})) = \mathbf{w}_r - \mathbf{x}_r] \\ \leq & 1/(2^n-1) \end{array} \quad (1)$$

Hence, H_r is ϵ-universal.

Furthermore, for any $\mathbf{y} \in \mathbb{B}^n$, we can construct $\mathbf{w}$ by choosing any $\mathbf{w}'_{r-1} \neq \mathbf{x}'_{r-1}$, and $\mathbf{w}_r = \mathbf{K}^{-1}\mathbf{y} + H_{r-1}(\mathbf{K}, \mathbf{w}'_{r-1})$. In this case, we have $H_r(\mathbf{K}, \mathbf{w}) = \mathbf{y}$ and $\mathbf{w} \neq \mathbf{x}$. From Equation 1, we have

$$\Pr[H_r(\mathbf{K}, \mathbf{x}) = \mathbf{y}] = \Pr[H_r(\mathbf{K}, \mathbf{x}) = H_r(\mathbf{K}, \mathbf{w})] \leq 1/(2^n-1)$$

Hence, H_r is also ϵ-balanced. □

Now we show how to make use of a nonzero initial state $\mathbf{s}_0$ to handle messages with only zero blocks and messages with different lengths, thereby completing the analysis. Essentially, hashing a message with a nonzero $\mathbf{s}_0$ is equivalent to prepending the nonzero block $\mathbf{K}^{-1}\mathbf{s}_0$.

Theorem 1. *For $n > 0$, $\epsilon = 1/(2^n-1)$, and nonzero $\mathbf{s}_0 \in \mathbb{B}^n$, the hash family $H : \mathcal{N} \times \mathbb{B}^{n*} \to \mathbb{B}^n$, defined as $H(\mathbf{K}, \mathbf{m}) = [\mathbf{K}^r\ \mathbf{K}^{r-1}\ \ldots\ \mathbf{K}]\mathbf{m} + \mathbf{K}^r\mathbf{s}_0$, where $r = |\mathbf{m}|_n$, is ϵ-balanced. That is, for any $\mathbf{x} \in \mathbb{B}^{n*}$ and $\mathbf{y} \in \mathbb{B}^n$, $\Pr[H(\mathbf{K}, \mathbf{x}) = \mathbf{y}] \leq \epsilon$. Furthermore, H is ϵ-universal. That is, for any $\mathbf{x}, \mathbf{w} \in \mathbb{B}^{n*}$, if $|\mathbf{x}| \neq |\mathbf{w}|$ or $\mathbf{x} - \mathbf{w} \neq \mathbf{0}$, then $\Pr[H(\mathbf{K}, \mathbf{x}) = H(\mathbf{K}, \mathbf{w})] \leq \epsilon$. Both probabilities are taken over random $\mathbf{K}$ uniformly chosen from $\mathcal{N}$.*

Proof. Since $\mathbf{K}$ is invertible, $\mathbf{v} = \mathbf{K}^{-1}\mathbf{s_0}$ is a nonzero block. For any r-block message $\mathbf{x}$, let $\widehat{\mathbf{x}} = (\mathbf{v}, \mathbf{x})$ be the $(r+1)$-block vector obtained by prepending $\mathbf{v}$ to $\mathbf{x}$, we have

$$\begin{array}{rcl} H(\mathbf{K}, \mathbf{x}) & = & [\mathbf{K}^r\ \mathbf{K}^{r-1}\ \ldots\ \mathbf{K}]\mathbf{x} + \mathbf{K}^{r+1}\mathbf{v} \\ & = & [\mathbf{K}^{r+1}\ \mathbf{K}^r\ \mathbf{K}^{r-1}\ \ldots\ \mathbf{K}](\mathbf{v}, \mathbf{x}) \\ & = & H_{r+1}(\mathbf{K}, \widehat{\mathbf{x}}) \end{array}$$

where H_{r+1} is as defined in Lemma 2.

Hence, for any $\mathbf{y} \in \mathbb{B}^n$, since $\widehat{\mathbf{x}} \neq \mathbf{0}$, from Lemma 2, we have

$$\Pr[H(\mathbf{K}, \mathbf{x}) = \mathbf{y}] = \Pr[H_{r+1}(\mathbf{K}, \widehat{\mathbf{x}}) = \mathbf{y}] \leq 1/(2^n-1) \quad (2)$$

Furthermore, without loss of generality, let $\mathbf{w} \in \mathbb{B}^{sn}$ be another s-block message where $1 \leq s \leq r$, such that either $\mathbf{w}$ is a shorter message (i.e., $s < r$), or $\mathbf{x}$ and $\mathbf{w}$ are of the same length ($s = r$) and $\mathbf{x} \neq \mathbf{w}$.

Now, let $\widehat{\mathbf{w}} = (\mathbf{0}, \ldots, \mathbf{0}, \mathbf{v}, \mathbf{w})$ be the result of prepending $r - s$ zero blocks (if $s < r$) and $\mathbf{v}$ to $\mathbf{w}$. In this case, $\widehat{\mathbf{x}}$ and $\widehat{\mathbf{w}}$ are of the same length $(r+1)n$, and $\widehat{\mathbf{x}} \neq \widehat{\mathbf{w}}$. Also, by construction $H(\mathbf{K}, \mathbf{w}) = H_{r+1}(\mathbf{K}, \widehat{\mathbf{w}})$. Therefore, from Lemma 2, we have

$$\begin{array}{cl} & \Pr[H(\mathbf{K}, \mathbf{x}) = H(\mathbf{K}, \mathbf{w})] \\ = & \Pr[H_{r+1}(\mathbf{K}, \widehat{\mathbf{w}}) = H_{r+1}(\mathbf{K}, \widehat{\mathbf{w}})] \leq 1/(2^n-1) \end{array} \quad (3)$$

Hence, the hash family H is ϵ-balanced and ϵ-universal as claimed. □

V. Implementation Considerations

A. Key Decomposition and Precomputation

For an n by n key matrix $\mathbf{K}$, and $n = st$ for some factors s and t, we can further divide it into s^2 sub-keys, which are sub-matrices of size t by t, as illustrated below.

$$\mathbf{K} = \begin{bmatrix} \mathbf{K}_{1,1} & \mathbf{K}_{1,2} & \ldots & \mathbf{K}_{1,s} \\ \mathbf{K}_{2,1} & \mathbf{K}_{2,2} & \ldots & \mathbf{K}_{2,s} \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{K}_{s,1} & \mathbf{K}_{s,2} & \ldots & \mathbf{K}_{s,s} \end{bmatrix}$$

For an n-bit data block $\mathbf{b}$, we similarly divide it into s sub-blocks of t bits each, $\mathbf{b} = (\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_s)$, and the matrix multiplication can be done by computing $\mathbf{K}_{i,j}\mathbf{b}_j$ for each $1 \leq i, j \leq s$.

To speed up the computation, we can pre-compute the values of $\mathbf{K}_{i,j}\mathbf{v}$ for each $\mathbf{K}_{i,j}$ and all possible values of the vector $\mathbf{v}$. The results can be grouped by $\mathbf{K}_{i,j}$ and stored as an array indexed by the integer value v of $\mathbf{v}$ when viewed as an t-bit integer. We denote such a pre-computed value as $\mathbf{K}_{i,j}[v]$. Subsequently, the computation $\mathbf{K}_{i,j}\mathbf{b}_j = \mathbf{K}_{i,j}[b_j]$ amounts to a simple table lookup.

The amount of memory required for the precomputed table is $s^2 2^t t = ns2^t$ bits. To perform a multiplication by $\mathbf{K}$ on a block $\mathbf{b}$, we need s^2 table look-ups and s^2 x-or operations. In comparison, a straightforward implementation of the multiplication takes n table look-ups and n x-or operations on the key columns. Therefore, the precomputed version may have an advantage when $s^2 < n$.

For example, if $n = 32$, we can choose $s = 4$ and the amount of memory required for the precomputation is $32 \cdot 4 \cdot 2^8 = 2^{15}$ bits, or 4 KB, and 16 table look-ups and 16 x-or operations are required to perform a multiplication by $\mathbf{K}$, compared to 32 table look-ups and 32 x-or operations when implemented without precomputation.

Furthermore, we note that the matrix decomposition method presented can also be applied without precomputation when the required hash size n is larger than the CPU register size. For example, to compute 128-bit hash values with only 32-bit registers, it would be natural to decompose the key matrix as the above where $s = 4$.

B. Key Generation

In our scheme we require the key $\mathbf{K}$ to be a uniformly random nonsingular Boolean matrix. We follow the approach in [8] and give a practical key generation algorithm. It is observed that random number generators in practice typically generate output in blocks of bytes instead of a stream of bits. Furthermore, it is a common practice to extend a short key by applying a key derivation function or a (non-cryptographic) random number generator algorithm. On the other hand, operations such as swapping data in the main memory can be expensive. Hence, it makes sense to optimize the speed instead of the use of randomness. It is also noted that matrices are stored as arrays of columns to speed up multiplications.

In particular, we propose the following algorithm to implement the method by Randall [8], where R denotes a random generator function from which we can draw k random bits by calling $R(k)$. Each n-bit vector is represented as an integer and the logical operations are bitwise operations on the integers. Each matrix is an n-by-n Boolean matrix, represented as an array of integers.

Essentially, we observe that the Randall method, when ignoring the amount of randomness used, is equivalent to the following steps: (1) generate a random unit upper triangular matrix $\mathbf{T}$, (2) generate a random unit lower triangular matrix $\mathbf{A}$, (3) generate n random numbers, say, $r_1, r_2, \ldots, r_n$, (4) for the i-th column of $\mathbf{A}$, permute it to the r_i-th column, and at the same time, for the r_i-th column of $\mathbf{T}$, clear the bits in the column except those that are in the locations $r_1, r_2, \ldots, r_i$. The Randall method avoids step (3) above by using the randomness already used to generate rows of $\mathbf{T}$. Instead, we explicitly generate these values. In this way, we can generate columns of $\mathbf{T}$ directly instead of generating the rows of $\mathbf{T}$ and then transposing the result. Furthermore, we use a bit mask to keep track of the rows already generated in $\mathbf{A}$, which allows iterative generation of $\mathbf{A}$ instead of recursively generating minor matrices. These optimizations allow fast key generation without affecting the distribution of the generated matrices. The resulting algorithm is shown as Algorithm 4, where $\ll$ denotes left bit shift of an integer, $\vee$ and $\wedge$ denote bit-wise logical-or and logical-and operations respectively.

Algorithm 4 Key Generation

```
1:  procedure GenKey($n$, $R$)
2:      $\mathbf{A} \leftarrow \mathbf{0}$. Write $\mathbf{A} = [a_1, \cdots, a_n]$ as an array of $n$-bit integers.
3:      $\mathbf{T} \leftarrow$ Random unit upper triangular matrix. Write $\mathbf{T} = [t_1, \cdots, t_n]$.
4:      $mask \leftarrow 0$
5:      for $i = 1, \ldots, n$ do
6:          $v \leftarrow 0$
7:          while $v = 0$ do
8:              $v \leftarrow R(n) \wedge \neg mask$
9:          end while
10:         $r \leftarrow$ The index of the first nonzero coordinate of $v$.
11:         $a_r \leftarrow R(n-i) \vee (1 \ll (n-i))$
12:         $mask \leftarrow mask \vee (1 \ll (n-r-1))$
13:         $t_r \leftarrow t_r \wedge mask$
14:     end for
15:     return $\mathbf{A} * \mathbf{T}$
16: end procedure
```

VI. Performance Evaluation

A. Key Generation

The performance evaluations of the proposed algorithms are done on a Ubuntu desktop PC with an x86_64 CPU running at 2.20 GHz. The algorithms are implemented in standard C, compiled using GCC with -O3 optimization.

For comparison, we also implemented a naïve key generation method, where we keep generating random Boolean matrices until we have a full rank matrix. We repeatedly generate n-by-n key matrices for 5000 times and record the average time it takes to generate one key matrix.

For $n = 32$, the naïve method takes 0.031 milliseconds in average to generate one key, whereas the proposed Algorithm 4 takes 0.010 milliseconds. For $n = 64$, the values are 0.090 milliseconds and 0.034 milliseconds respectively. In other words, the proposed algorithm of implementing the Randall method takes roughly one third of the time taken by the naïve method. Considering that the probability that a random Boolean matrix is nonsingular is slightly less than 0.3 (as observed in [8]), the proposed algorithm is near-optimal in practice.

B. Serial Hashing

To avoid complicating the evaluation process, we only implement the Basic Hash as in Algorithm 1 in a single thread,

TABLE I
AVERAGE PROCESSING TIME PER MB (SINGLE PASS, $n = 32$)

Algorithm	B=10	20	50	100	200
Basic Hash	29.8 (ms)	18.4	12.5	10.8	9.9
SHA1	24.3	13.0	6.4	4.3	3.2
SHA256	31.6	19.7	11.1	8.3	6.6

TABLE II
AVERAGE PROCESSING TIME PER MB (SINGLE PASS, $n = 64$)

Algorithm	B=10	20	50	100	200
Basic Hash	15.5 (ms)	9.6	6.8	5.8	5.4
SHA1	14.6	7.4	4.4	3.2	2.7
SHA256	22.8	12.2	8.4	6.6	6.0

and compare its performance with SHA1 and SHA256 implementation from the stock mbedTLS package from Ubuntu 20.04. It should be noted that a major advantage of our scheme is the ability of flexible parallel processing, and the result presented here shows its performance in a single thread only. Further performance gain is expected for a multi-threaded or distributed implementation.

The tests are done by repeatedly hashing messages of different sizes and gathering the average processing time per megabyte (MB) of data. The messages are randomly generated and are always multiples of the block size n. In the tables below, B represent the number of blocks in the messages, and the entries represent the average time (in milliseconds) it takes to hash 1 MB.

In Table I and II, we first present the results where, for each test, the entire message is fed to the hash function at once. In other words, there is only a single pass for each message.

For memory integrity protection, it would make more sense to compare the performance where the input messages are fed to the hash function block by block (incremental hashing), instead of sending the entire message at the same time. Unsurprisingly, the proposed hash function exhibits much better comparative performance, as illustrated in Table III and IV.

TABLE III
AVERAGE PROCESSING TIME PER MB (INCREMENTAL HASHING, $n = 32$)

Algorithm	B=10	20	50	100	200
Basic Hash	198.2 (ms)	178.9	168.9	167.5	165.4
SHA1	231.8	192.7	171.7	166.7	163.1
SHA256	240.4	199.6	176.7	171.5	166.7

TABLE IV
AVERAGE PROCESSING TIME PER MB (INCREMENTAL HASHING, $n = 64$)

Algorithm	B=10	20	50	100	200
Basic Hash	105.8 (ms)	95.9	91.8	89.9	88.8
SHA1	119.4	96.7	88.3	84.0	81.4
SHA256	127.5	101.8	93.4	89.1	86.2

As can be seen from these tables, the throughput of the hash functions become higher for longer messages in general. In single pass hashing (Table I and II), the performance of the proposed Basic Hash is comparable to SHA1 and SHA256 for short messages. However, in the incremental hashing setup (Table III and IV), the proposed hash function clearly outperforms both SHA1 and SHA256 for short messages, and performs equally well for long messages. Furthermore, as mentioned earlier, further performance gain is expected when hashing is done in parallel.

VII. CONCLUSIONS

In this paper, we propose an ϵ-balanced (and ϵ-universal) hash family, where the key size is n^2 and the bound $\epsilon = 1/(2^n - 1)$, which only depends on the hash size n, and does not increase with the message size as in previously work. The proposed hash family is also easily parallelized and the division of the input message can be arbitrary. The parameter n is also the input block size and internal state size, which can be chosen arbitrarily to match the required CPU and memory architectures and security requirements without redesign. The simple mathematical structure of the scheme not only allows for rigorous information-theoretic proofs but also efficient software and hardware implementations with high performance. The proposed hash family can be used as a building block to construct secure message authentication codes, and is particularly suitable for memory integrity protection.

REFERENCES

[1] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In *Advances in Cryptology - CRYPTO*, volume 4117 of *LNCS*, pages 602–619, 2006.

[2] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology - CRYPTO*, volume 1109 of *LNCS*, pages 1–15, 1996.

[3] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In *Advances in Cryptology - CRYPTO*, volume 1666 of *LNCS*, pages 216–233, 1999.

[4] L. Carter and M. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[5] Hugo Krawczyk. LFSR-based hashing and message authentication. In *Advances in Cryptology - CRYPTO*, volume 839 of *LNCS*, pages 129–139, 1994.

[6] Hugo Krawczyk. New hash functions for message authentication. In *Advances in Cryptology - EUROCRYPT*, volume 921 of *LNCS*, pages 301–310, 1995.

[7] David A. McGrew and John Viega. The security and performance of the galois/counter mode (GCM) of operation. In *Progress in Cryptology - INDOCRYPT*, volume 3348 of *LNCS*, pages 343–355, 2004.

[8] Dana Randall. Efficient generation of random nonsingular matrics. *Random Structures and Algorithms*, 1993.

[9] Victor Shoup. On fast and provably secure message authentication based on universal hashing. In *Advances in Cryptology - CRYPTO*, volume 1109 of *LNCS*, pages 313–328, 1996.