Fast Universal Hashing with Small Keys and No Preprocessing: The PolyR Construction

January 4, 2001

Abstract

We describe a universal hash-function family, PolyR, which hashes messages of effectively arbitrary lengths in 3.9–6.9 cycles/byte (cpb) on a Pentium II (achieving a collision probability in the range 2^{-16}–2^{-50}). Unlike most proposals, PolyR actually hashes short messages faster (per byte) than long ones. At the same time, its key is only a few bytes, the output is only a few bytes, and no "preprocessing" is needed to achieve maximal efficiency. Our designs have been strongly influenced by low-level considerations relevant to software speed, and experimental results are given throughout.

Keywords: Universal hashing, software-optimized hashing, message authentication.

1 Introduction

Ever since its introduction by Carter and Wegman in 1979 [6], universal hashing has been an important tool in computer science. Recent attention has been paid to universal hashing as a method to authenticate messages, an idea also proposed by these authors [12]. Its use in authentication has resulted in several very fast universal hash functions with low collision probabilities. But the implementations of these fastest universal hash functions tend to require either significant precomputed data, long keys or special-purpose hardware to achieve their impressive speeds.

Our contribution is a polynomial-based hash function we call PolyR. This hash function is not as fast as the fastest hash functions which have been designed for message authentication—speed is about 3.9–6.9 cpb. But that is still very fast, and, compared to the fastest of hash functions, PolyR has some different and desirable characteristics. First, it hashes messages of essentially *any* length (and varying lengths are fine). The key is short (say 28 bytes), independent of the message length. The key requires no preprocessing: the natural representation of the key is the desirable one for achieving good efficiency. Quite pleasantly, the hash function is fastest, per byte, on *short* messages—it actually gets slower, per byte, as the message gets longer (the rates are constant until particular threshold lengths are crossed, like 2^{11} and 2^{33} bytes). This is the exact reverse of most optimized hash functions having short output lengths: they do better as the message gets longer. If used for authentication, working best for short messages is desirable insofar as *most* network traffic is short. Finally, implementation of our hash function family is simple and requires no special hardware (like floating-point units or multimedia execution units) to do well.

The hash function family PolyR was designed for use in a multi-layer hashing construction, to be used for fast message authentication. In such constructions a very fast first layer of hashing is applied to an incoming message to compress it to a small fraction of its original length. This compressed message is then passed to PolyR. When used as a second hash-layer in this manner, it

Hash Function	Collision Bound	Code + Data Size	Speed (cpb)	Output (bits)
This paper	$n2^{-28}$	$(124+8)$ bytes	3.9	32
This paper	$n2^{-49}$	$(409+16)$ bytes	6.9	64
Division Hash [10]	$n2^{-59}$	$\sim(?+8)$ KB	7.5	64
UHASH-16 [3]	2^{-60}	$\sim(7+2)$ KB	1.0	64
UHASH-32 [3]	2^{-60}	$\sim(8+2)$ KB	2.0	64
hash127 [2]	$n2^{-127}$	$\sim(4+1.5)$ KB	4.3	127
MD5	unknown	1.7 KB	5.3	128
SHA-1	unknown	4.3 KB	13.1	160

Table 1: *Comparing the new constructions with some other hash families. Sizes marked with "$\sim$" are conservative estimates. All timings are for the fastest Pentium/Pentium II timings reported. To obtain smaller collision bounds one can hash twice or use the methods of this paper with $p(96)$ or $p(128)$.*

can be expected that the *vast* majority of messages fed to PolyR will be short, since messages must be quite huge indeed before the second-layer compressed message gets long.

The hash function PolyR is a refinement to the classical suggestion of Carter and Wegman where one treats the message as specifying the coefficients of a polynomial, and one evaluates that polynomial at a point which is the key. Our refinements involve: (1) choosing the base field to be a prime just smaller than a power of 2^{32}, 2^{64}, or 2^{128} (this is a common trick); (2) using a simple "translation" trick to take care of the problem that some messages will now give rise to coefficients not in the field (because our field is just smaller than a power of two); (3) limiting the key space to a particular "convenient" subset of all the field points; and (4) using a "ramping-up" trick so that we don't have to pay in efficiency for short messages in order that the method can handle long ones. The result is a simple, flexible, fast-to-compute hash function. These various tricks, individually rather modest, work together to rise to a quite a nice hash-function family.

1.1 Related Work

Carter and Wegman introduced the ideas of universal hashing and using polynomials in universal hashing in 1979 [6]. Since that time polynomials have been used for fast hashing in many other works. In his "Cryptographic CRC", Krawczyk views messages to be hashed as polynomials over GF(2) which are divided modulo a random irreducible polynomial [8]. The division can be done quickly in hardware using linear feedback shift registers. Shoup describes several variants of polynomial hashing and provides implementation results [10]. His "generalized division hash", which bounds collisions between $64n$-bit messages as no more than $n2^{-59}$, views messages as polynomials over GF(2^{64}), uses an 8 KB precomputed table, and has a throughput of 7.5 cpb on a Pentium [10]. Afanassiev, Gehrmann and Smeets discuss fast polynomial hashing modulo random 3-, 4- and 5-nomials [1]. Their methods use small keys, but no implementation results are provided. Bernstein defines *hash127*, a polynomial hash defined over a large prime field, $\mathbb{Z}_{p(127)}$ [2]. Over $32n$-bit messages it has a collision probability of no more than $n2^{-127}$. Bernstein's implementation uses floating-point operations and a 1.5 KB precomputed table to achieve a throughput of 4.3 cpb on a Pentium II.

Other software efficient universal hash functions include Rogaway's bucket hash [9]; the MMH function of Halevi and Krawczyk [7]; and the NH function of Black, Halevi, Krawczyk, Krovetz and Rogaway [4]. The last is the current speed champion, providing collision probabilities of 2^{-60} with

4 KB of precomputed data and achieving throughput of 1.0 cpb on a Pentium II. using Pentium MMX instructions, and 1.9 cpb without MMX.

If one does not require the combinatoric certainties of universal hashing, one could employ cryptographic hashing to construct hash functions with short output lengths, short keys and little preprocessing. Bosselaers, Govaerts and Vandewalle report on optimized Pentium timing for several cryptographic hash functions: MD4 (3.8 cpb), MD5 (5.3 cpb) and SHA-1 (13.1 cpb) [5]. Simple methods can be used to convert these function into universal hash functions by, for example, keying their initial values [11]. We do not know what the collision probability would be for such constructions; for such a transformation to result in a good universal hash function, certain unproved assumptions must be made about the cryptographic hash function.

1.2 Notation

The algorithms described in this paper manipulate both bit-strings and integers. The i-th bit of string M is denoted $M[i]$ (bit-indices begin with 1). The substring consisting of the i-th through j-th bits of M is denoted $M[i \ldots j]$. The concatenation of string M_1 followed by string M_2 is denoted $M_1 \parallel M_2$. The length in bits of string M is $|M|$. The string of n zero-bits is denoted 0^n.

Given $b > 1$, the constant $p(b)$ is the largest prime smaller than 2^b. Given string M and $b > 0$, $\mathsf{padonezero}(M, b)$ returns the string $M \parallel 1 \parallel 0^n$, where n is the smallest number that makes the length of $M \parallel 1 \parallel 0^n$ divisible by b. Given a string M, the function $\mathsf{str2num}(M)$ returns the integer that results when M is interpreted as an unsigned binary number. Similarly, $\mathsf{num2str}(n, b)$ produces the unique b-bit string which is the binary representation for the non-negative number n.

The number of elements in a set S is denoted $|S|$.

1.3 Organization

In the next few sections we develop a fast polynomial hash function. We build up to it in a couple of stages. In the appendix we generalize the hash function using arbitrary parameters. Theorems are given in both cases, but proven only for the concrete case. Proofs for the parameterized cases are straightforward adaptations of the ones for the concrete version, so they are omitted. Understanding the algorithms, theorems and proofs is easier in the concrete examples.

2 Carter-Wegman Polynomial Hashing: PolyCW

We begin by reviewing the "standard" approach for polynomial hashing. Let $\mathbb{F}$ be a finite field, let $k \in \mathbb{F}$ be a point in that field (the "key") and let $\mathbf{m} = (m_0, \ldots, m_n)$ be a vector of points in $\mathbb{F}$ that we want to hash (the "message"). We can hash message $\mathbf{m}$ to a point y in $\mathbb{F}$ (the "hash value") by computing $y = m_0 k^n + \cdots + m_{n-1} k^1 + m_n k^0$, where all arithmetic is done in $\mathbb{F}$. We denote this family of hash functions as $\mathsf{PolyCW}[\mathbb{F}]$. The computation of this hash function (with $n+1$ multiplications in the field and $n+1$ additions in the field) is described in Figure 1.[1]

$\mathsf{PolyCW}[\mathbb{F}]$ is one of the most well-known universal hash-function families. It was described by Carter and Wegman in the paper that introduced that notion [6]. The main property it has is as follows. If $\mathbf{m} = (m_n, \ldots, m_0)$ and $\mathbf{m}' = (m'_n, \ldots, m'_0)$ are distinct vectors with the same number of components then $\Pr[H \leftarrow \mathsf{PolyCW}[\mathbb{F}]\,;\, k \xleftarrow{R} \mathbb{F} \,:\, H_k(\mathbf{m}) = H_k(\mathbf{m}')] \leq \frac{n}{|\mathbb{F}|}$. This result is due to the Fundamental Theorem of Algebra which states that a nonzero polynomial of degree at most

[1] All algorithms depicted in this paper which evaluate polynomials do so by using Horner's Rule which says that polynomial $m_0 k^n + \cdots + m_{n-1} k^1 + m_n k^0$ can be rewritten as $m_n + k(m_{n-1} + k(m_{n-2} + k(m_{n-3} + \cdots)))$. This allows for simple iteration with one multiplication and one addition for each element of the message.

```
algorithm PolyCW [𝔽](k, m)
// Parameter: 𝔽 is a finite field.
// Input: k ∈ 𝔽 and m = (m_0, ..., m_n) where m_i ∈ 𝔽 for 0 ≤ i ≤ n.
// Output: y ∈ 𝔽.
Let n be the number of elements in m
y = 0
for i ← 0 to n do
    y ← ky + m_i                          // Arithmetic in 𝔽
return y
```

Figure 1: *The basic polynomial-hashing method of Carter and Wegman on which we build. The message* $\mathbf{m} = (m_0, \ldots, m_n)$ *is hashed to* $\sum_{i=0}^{n} m_i k^{n-i}$.

```
algorithm PolyP32(k, m)
// Input: k ∈ K_32 and m = (m_1, ..., m_n) where m_i ∈ Z_{p(32)} for 1 ≤ i ≤ n.
// Output: y ∈ Z_{p(32)}.
Let n be the number of elements in m
p ← 2^32 − 5                              // The largest prime smaller than 2^32
y = 0
for i ← 1 to n do
    y ← ky + m_i mod p
return y
```

Figure 2: *The* PolyP32 *algorithm. A variant of the* PolyCW *hash, accelerated by choosing a field* $\mathbb{Z}_{p(32)}$ *in which calculations can be performed quickly and choosing a key-set* K_{32} *which reduces arithmetic overflow on 32-bit processors. The* **for** *loop could be rewritten as the polynomial:* $y = \sum_{i=1}^{n}(m_i k^{n-i}) \bmod p$.

n can have at most n roots. Rewriting the above probability as $\Pr[k \xleftarrow{R} \mathbb{F} : \sum_{i=0}^{n} m_i k^{n-i} = \sum_{i=0}^{n} m_i k^{n-i}] = \Pr[k \xleftarrow{R} \mathbb{F} : \sum_{i=0}^{n}(m_i - m'_i)k^{n-i} = 0]$, and applying the Fundamental Theorem, we see that there can be at most n values for k which cause $\sum_{i=0}^{n}(m_i - m'_i)k^{n-i}$ to evaluate to zero.

3 Making PolyCW [𝔽] Fast

Care must be taken in the implementation of PolyCW [𝔽]. A naive implementation is unlikely to perform well. Many choices of 𝔽 and the set from which the hash-key is chosen can result in sub-optimal performance. We investigate the effect that shrewd choices for 𝔽 and the key-set have on performance.

Field Selection. To make an efficient and practical hash function out of PolyCW [𝔽] we should carefully choose the finite field 𝔽. Fields like GF[2^{64}] make natural candidates, because we are ultimately interested in hashing bit strings which are easily partitioned into 64-bit substrings. But

arithmetic in GF[2^w] turns out to be less convenient for contemporary CPUs than a well-chosen alternative. In this paper we will do better by using prime fields in which the prime is just smaller than a power of two.

Consider first the use of the prime $p(32) = 2^{32} - 5$, which is the largest prime less than 2^{32}. To implement PolyCW $[\mathbb{Z}_{p(32)}]$ efficiently, we need a good way to calculate $y \leftarrow ky + m \bmod p(32)$, where $y, k, m \in \mathbb{Z}_{p(32)}$. There are several options. One's first instinct is to use the native "mod" operand of a high-level programming language (like "%" in C), or to use a corresponding operator in the hardware architecture. But these choices are usually slow. For example, PolyCW $[\mathbb{Z}_{p(32)}]$, implemented in assembly using the native mod operator runs in **12.4** cpb (cycles/byte) on a Pentium II.

A faster method exploits the fact that since $p(32) = 2^{32} - 5$, the numbers 2^{32} and 5 are equivalent in the field $\mathbb{Z}_{p(32)}$, so $2^{33} = 10$, $2^{34} = 20$ and, more generally, $a2^{32} = 5a$ in $\mathbb{Z}_{p(32)}$. So, to calculate $ky \bmod p(32)$, first compute the 64-bit product $z = ky$ and separate z into a 32-bit high-word a and a 32-bit low-word b so that $z = a2^{32} + b$. We can then use the observation just made and rewrite $z \bmod p(32)$ as $5a + b$. This means that the calculation $y = ky + m \bmod p(32)$ can be done by computing $y = 5a + b + m \bmod p(32)$, which can be implemented more cheaply than the original approach because it does not require division to perform the modular reduction.

Key-Set Selection. When implemented on a 32-bit architecture, the values a, b and m just discussed fit conveniently into 32-bit registers, making these quantities easy to manipulate. On most such architectures, the calculation of y is going to be fastest if it is done with minimal register overflow. To calculate $y = 5a + b + m \bmod p(32)$ using only 32-bit registers, we need one multiplication, two additions and then some additional instructions to handle register overflow. Each operation that can result in register overflow requires several instructions, including a conditional move or branch, to check and deal with the potential overflow event. To accelerate the calculation of y we reduce the number of potential overflows. Little can be done about overflow from the additions because both b and m can be nearly 2^{32}, but overflow from the multiplication can be eliminated. Only if a is larger than $\lfloor 2^{32}/5 \rfloor \approx 2^{29.7}$ can the term $5a$ overflow a 32-bit register. We can restrict a to safe values by restricting k to values less than 2^{29}. This allows for a faster implementation. The expense for this optimization is a higher collision probability because the key is chosen from a set of 2^{29} elements instead of a set of 2^{32} elements.

Divisionless modular reduction. Another optimization over a naive implementation is the elimination of division to calculate modular reductions. This technique is not new. In calculating $y = 5a + b + m \bmod p(32)$, each of the $5a$, b and m terms are less than 2^{32}. As we sum them using computer arithmetic with 32-bit registers, we can easily detect 32-bit overflows. Each such overflow indicates a 2^{32} term which is not accounted for in the resulting register. But, because $2^{32} \equiv 5$, these overflows are easily accounted for by adding 5 for each overflow to the resulting register. Done carefully, this observation results in a number y, derived without any division, which is representable in 32-bits (ie. $0 \le y < 2^{32}$). See Figure 3 for implementation details. Do we then need to reduce y to a number in $\mathbb{Z}_{p(32)}$? No. All of the discussion so far requires only that y be representable in a 32-bit register. Instead of reducing y to be in $\mathbb{Z}_{p(32)}$ after every intermediate calculation, we defer all such reductions until the end, when a final single reduction is performed.

Speed. Taken together, the selection of a convenient prime field and the restriction of the key-set to keys which eliminate some register overflows allows a nice speed-up over a naive implementation of PolyCW. Figure 2 shows a version of polynomial hash based upon PolyCW which hashes over the field $\mathbb{Z}_{p(32)}$ and restricts key selection to the set $K_{32} = \{a : 0 \le a < 2^{29}\}$. Our implementation of the core $y = ky + m \bmod p(32)$ calculation uses just 8 lines of Pentium II assembly (Figure 3)

```
; Calculate y = y * k + m mod p(32)
; Assume y is in register eax before and after code segment.
mul    k                          ; edx:eax = k * y
lea    edx, [edx*4+edx]           ; edx = 5 * edx
add    eax, edx                   ; eax = edx + eax
lea    edx, [eax+5]               ; edx = eax + 5
cmovc  eax, edx                   ; if (carried) then eax = edx
add    eax, m                     ; eax = eax + m
lea    edx, [eax+5]               ; edx = eax + 5
cmovc  eax, edx                   ; if (carried) then eax = edx
```

Figure 3: *The $y = ky + m$ calculation of the* PolyP32 *algorithm written in Pentium II assembly. The flag "*`carried`*" is true only if the previous* `add` *instruction causes a register overflow. The conditional-move instruction (*`cmovc`*) is used to avoid any branches during execution of the routine, and the load-effective-address instruction (*`lea`*) is used for addition and multiplication of small constants. The result of the routine could possibly be in the range $p \leq y < 2^{32}$, which is outside of the field $\mathbb{Z}_{p(32)}$, but this is easily fixed with a single subtraction after hashing the final word of the entire message.*

and achieves a peak throughput of **3.69** cpb.

We state here the (simple) proposition establishing the collision bound of the PolyP32 hash function.

Proposition 3.1 *For any positive integer n and any distinct messages $\mathbf{m} = (m_0, \ldots, m_n)$ and $\mathbf{m}' = (m'_0, \ldots, m'_n)$, consisting of elements from $\mathbb{Z}_{p(32)}$, $\Pr[k \leftarrow K_{32} : \mathsf{PolyP32}(k, \mathbf{m}) = \mathsf{PolyP32}(k, \mathbf{m}')] \leq n/|K_{32}| = n2^{-29}$.*

64-Bit Hashing and Key Restriction. We also implemented an analogous PolyP64 hash function whose core calculation is $y = ky + m \bmod p(64)$ where $p(64) = 2^{64} - 59$ and k, y and m are all elements of $\mathbb{Z}_{p(64)}$. As in the 32-bit case, it is cheapest to calculate the result without using division. If we let $2^{32}k_h + k_\ell$ represent k and $2^{32}y_h + y_\ell$ represent y, then ky can be calculated as $ky = 2^{64}k_h y_h + 2^{32}(k_h y_\ell + k_\ell y_h) + k_\ell y_\ell$. Again, restricting the set of values that k can take on allows for faster implementations by eliminating some 32-bit register overflows. We define key-set $K_{64} = \{a2^{32} + b : 0 \leq a, b < 2^{25}\}$. This restriction allows an implementation of PolyP64 which has a collision probability of $(n/2^{50})$, uses 40 lines of assembly and has a peak throughput of **6.86** cpb.

4 Expanding the Domain to Arbitrary Strings

The hash function PolyP32 is not generally useful. It only works on same-length messages, and those messages must be made of elements from the field $\mathbb{Z}_{p(32)}$. We now remove these limitations and develop PolyQ32. The result, depicted in Figure 4, hashes most messages at a rate of **3.86** cpb.

Allowing Variations in Length. It is a trivial exercise to produce two different-length messages which collide when hashed with PolyP32 using *any* key: under PolyP32, the hash of a message $\mathbf{m} = (m_0, \ldots, m_n)$ using key k is simply $h(k, \mathbf{m}) = m_0 k^n + \cdots + m_n k^0 \bmod p(32)$, so prepending 0

```
algorithm PolyQ32(k, M)
// Input: k ∈ K_32 and M ∈ ({0,1}^32)^+.
// Output: y ∈ Z_{p(32)}.
p ← 2^32 − 5                          // The largest prime smaller than 2^32
offset ← 5                            // Constant for translating out-of-range words
marker ← 2^32 − 6                     // Constant for indicating out-of-range words
n ← |M|/32
M_1 ‖ ... ‖ M_n ← M,                  // Break M into 32-bit chunks
     where |M_1| = ··· = |M_n| = 32
y ← 1                                 // Set highest coefficient to 1
for i ← 1 to n do
     m ← str2num(M_i)
     if (m ≥ p − 1) then              // If word is not in range, then
          y ← ky + marker mod p       // Marker indicates out-of-range
          y ← ky + (m − offset) mod p // Offset m back into range
     else
          y ← ky + m mod p            // Otherwise hash in-range word
return y
```

Figure 4: *The* PolyQ32 *algorithm. The* PolyP32 *hash extended to hash strings instead of vectors of field elements and to allow good collision probabilities over two strings which differ in length.*

to the vector $\mathbf{m}$ results in a message $\mathbf{m}' = (0, m_0, \ldots, m_n)$ which is hashed as $h(k, \mathbf{m}') = 0k^{n+1} + m_0k^n + \cdots + m_nk^0 \bmod p(32)$ and is equal to $h(k, \mathbf{m})$ because the additional zero-term has no effect on the hash value. For the Fundamental Theorem of Algebra to guarantee a low number of roots (and hence a low collision probability), it is essential that the difference between $\mathbf{m}$ and $\mathbf{m}'$ be non-zero. This means that if the two vectors differ only in length, then at least one of the initial elements of the longer vector must be non-zero. To guarantee this we employ a standard trick and implicitly prepend a "1" to the vectors being hashed. Thus, the hash of $\mathbf{m} = (m_0, \ldots, m_n)$ implicitly becomes the hash of $\mathbf{m} = (1, m_0, \ldots, m_n)$, and the hash of $\mathbf{m}' = (0, m_0, \ldots, m_n)$ implicitly becomes the hash of $\mathbf{m}' = (1, 0, m_0, \ldots, m_n)$. The difference between these two vectors is non-zero. The following theorem assures that augmenting PolyP32 in this way results in a hash with nearly the same collision probability as PolyP32, but works over messages of different lengths.

Proposition 4.1 *Let $\ell < n$ be positive integers. Let $\mathbf{m} = (m_0, \ldots, m_\ell)$ and $\mathbf{m}' = (m'_0, \ldots, m'_n)$ be any two vectors of elements from the field $\mathbb{F}$. Then there are at most $n+1$ values for $k \in \mathbb{F}$ such that $k^{\ell+1} + \sum_{i=0}^{\ell} m_i k^{\ell-i} = k^{n+1} + \sum_{i=0}^{n} m'_i k^{n-i}$.*

Proof. Beginning with $k^{\ell+1} + \sum_{i=0}^{\ell} m_i k^{\ell-i} = k^{n+1} + \sum_{i=0}^{n} m'_i k^{n-i}$, and moving all of its terms to the right side of the equation we get $0 = k^{n+1} - k^{\ell+1} + \sum_{i=0}^{n} m'_i k^{n-i} - \sum_{i=0}^{\ell} m_i k^{\ell-i}$. But, the right side of this equations is now a non-zero polynomial, is of degree $n+1$, and therefore has at most $n+1$ roots. $\diamondsuit$

Alternative Method. Another method of augmenting PolyP32 to allow variable length messages is to use a second key $k' \in \mathbb{Z}_{p(32)}$ and add it to each element of the message being hashed. Thus,

$h(k, k', \mathbf{m})$ would be computed as $\sum_{i=0}^{\ell}(m_i + k')k^{\ell-i}$. This method requires an extra addition per message word being hashed and so the first method seems favorable.

ALLOWING BIT-STRINGS. To make the function PolyP32 of Figure 2 more useful, it must be adapted to allow bit-strings rather than only vectors from $\mathbb{Z}_{p(32)}$. The field $\mathbb{Z}_{p(32)}$ was chosen because it contains nearly all the numbers representable as 32-bit strings. Thus, when we desire to hash a bit-string, we may partition the string into 32-bit words and treat the partition as a vector of 32-bit numbers. PolyP32 can then hash the vast majority of the vector's elements without any modification. But, some of the 32-bit numbers may be in the range $p(32) \ldots 2^{32} - 1$, outside $\mathbb{Z}_{p(32)}$. What should be done with them?

One approach is to transform a vector of 32-bit numbers, which may have some elements outside of $\mathbb{Z}_{p(32)}$, into a vector which does not. The transformation must map distinct vectors into distinct vectors.

We solve this problem by examining a vector of 32-bit numbers and replacing each vector element m_i that is greater than $p(32) - 2$ with *two* numbers, $p(32) - 1$ and $m_i - 5$. Note that both of these numbers are in $\mathbb{Z}_{p(32)}$. Each such replacement lengthens the resulting vector by one element. Thus, the vector $\mathbf{m} = (4, 2^{32} - 3, 10)$, whose second element is greater than $p(32) - 2$, would be transformed into the vector $\mathbf{m'} = (4, 2^{32} - 6, 2^{32} - 8, 10)$. We call this transformation $\mathsf{DoubleTransform} : (\mathbb{Z}_{2^{32}})^+ \to (\mathbb{Z}_{p(32)})^+$. The following proposition assures that DoubleTransform is correct.

Proposition 4.2 *For positive integers ℓ and n, and distinct messages $\mathbf{m} = (m_0, \ldots, m_\ell)$ and $\mathbf{m}' = (m'_0, \ldots, m'_n)$ made of elements from $\mathbb{Z}_{2^{32}}$, the vectors $\mathsf{DoubleTransform}(\mathbf{m})$ and $\mathsf{DoubleTransform}(\mathbf{m}')$ consist of elements from $\mathbb{Z}_{p(32)}$ and are distinct.*

Proof. Let ℓ and n be positive integers and let $\mathbf{m} = (m_0, \ldots, m_\ell)$ and $\mathbf{m}' = (m'_0, \ldots, m'_n)$ be distinct vectors consisting of elements from $\mathbb{Z}_{2^{32}}$. Let $\mathbf{t} = \mathsf{DomainTransform}(\mathbf{m})$ and $\mathbf{t}' = \mathsf{DomainTransform}(\mathbf{m}')$. Let i be the smallest number such that $m_i \neq m'_i$. If such an i does not exist then one of $\mathbf{m}$ or $\mathbf{m}'$ must be a proper prefix of the other. In this case, any lengthening of the shorter vector by DoubleTransform must be mirrored by the transformation of the longer vector ensuring that the two remain different lengths after transformation.

If m_i and m'_i are both less than $p(32)-1$, then after transformation $t_i = m_i$ and $t'_i = m'_i$, ensuring that $\mathbf{t} \neq \mathbf{t}'$. If only one of m_i and m'_i is less than $p(32) - 1$, say m_i, then after transformation $t_i = m_i$ and $t'_i = p(32) - 1$, again ensuring that $\mathbf{t} \neq \mathbf{t}'$. Finally, if both m_i and m'_i are greater than $p(32) - 2$, then after transformation $t_{i+1} = m_i - 5$ and $t'_{i+1} = m_i - 5'$ again ensuring that $\mathbf{t} \neq \mathbf{t}'$. $\diamondsuit$

ALTERNATIVE METHOD. There are many ways to patch PolyP32 to allow out-of-range elements. One probabilistic alternative is to offset every out-of-range number by a randomly chosen $k' \in \{5, \ldots, 2^{32} - 5\}$. All out-of-range numbers are in $\{2^{32} - 5, \ldots, 2^{32} - 1\}$, so k', when subtracted from an out-of-range number, will always yield a number in $\mathbb{Z}_{p(32)}$. This method has the advantage of not increasing message length upon transformation, but requires an extra key element, and in practice does not speed hashing with respect to the method of Proposition 4.2.

Together, Propositions 4.1 and 4.2 prove the following corollary.

Corollary 4.3 *For any positive integers $\ell \leq n$ and distinct messages $M \in \{0,1\}^{32l}$ and $M' \in \{0,1\}^{32n}$, $\Pr_{k \in K_{32}}[\mathsf{PolyQ32}(k, M) = \mathsf{PolyQ32}(k, M')] \leq 2n/|K_{32}| = n2^{-28}$.*

The discussion so far has focussed on PolyQ32, a hash function defined on 32-bit words. An analogous 64-bit variant, PolyQ64, yields the following bound.

algorithm PolyR32_64($\mathbf{k}, M$)
// *Input:* $\mathbf{k} = (k_1, k_2)$ *with* $k_1 \in K_{32}$ *and* $k_2 \in K_{64}$, *and* $M \in \{0,1\}^*$.
// *Output:* $Y \in \{0,1\}^{64}$.
if ($|M| \leq 2^{14}$) **then** // 2^9 32-*bit words*
    $M \leftarrow$ padonezero($M, 32$)
    $y \leftarrow$ PolyQ32(k_1, M) // *Hash in* $\mathbb{Z}_{p(32)}$
else if ($|M| \leq 2^{36}$) **then** // 2^{30} 64-*bit words*
    $M_1 \leftarrow M[1 \ldots 2^{14}]$
    $M_2 \leftarrow M[2^{14}+1 \ldots |M|]$
    $M_2 \leftarrow$ padonezero($M_2, 64$)
    $y \leftarrow$ PolyQ32(k_1, M_1) // *Hash in* $\mathbb{Z}_{p(32)}$
    $y \leftarrow$ PolyQ64(k_2, num2str($y, 64$) $\|$ M_2) // *Hash in* $\mathbb{Z}_{p(64)}$, *prepending* y
else
    return Error // *Message too long*
$Y \leftarrow$ num2str($y, 64$) // *Convert to string*
return Y

Figure 5: *The* PolyR32_64 *algorithm. Combining the* PolyP32 *and* PolyP64 *hashes into a hash function which is fast on short messages but also performs well on long ones.* PolyR32_64 *also extends the domain to messages which are not a multiple of the constituent hashes word-lengths.*

Corollary 4.4 *For any positive integers* $\ell \leq n$ *and distinct messages* $M \in \{0,1\}^{64l}$ *and* $M' \in \{0,1\}^{64n}$, $\Pr_{k \in K_{64}}[\mathsf{PolyQ64}(k, M) = \mathsf{PolyQ64}(k, M')] \leq 2n/|K_{64}| = n2^{-49}$.

Two things are worth noting. First, the factor of two introduced in the $2n/|K_{32}|$ term is due to the potential doubling of message length by the DoubleTransform function. And, second, standard message padding techniques are not addressed in this paper. It is assumed that messages being hashed have been properly padded to a 32-bit boundary.

It should also be noted that the probability that PolyQ32 or PolyQ64 hash any message to a particular result is also low. Consider a message made of n 32-bit words $\mathbf{x} = (x_1, \ldots, x_n)$ and a constant c. If $c \geq p(32)$ then PolyQ32($k, \mathbf{x}$) cannot hash to c, and if $c < p(32)$, then PolyQ32($k, \mathbf{x}$) will hash to c only if PolyQ32($k, \mathbf{x}'$) hashes to zero where $\mathbf{x}' = (x_1, \ldots, x_n - c)$. After the DoubleTransform transformation of $\mathbf{x}'$, the Fundamental Theorem tells us that there are no more than $2n$ keys which allow this to happen.

Claim 4.5 *Let* n *and* c *be numbers, and message* $M \in \{0,1\}^{32\ell}$, $\Pr_{k \in K_{32}}[\mathsf{PolyQ32}(k, M) = c] \leq 2n/|K_{32}|$.

5 PolyR: Overcoming Polynomial Hash Length Limitations

Taking a closer look at the bounds established for each of the polynomial hash functions, one can see that the collision bounds degrade linearly along with the length of the messages being hashed. This is a byproduct of the use of polynomials in hashing: As messages get longer, so do the degrees of the polynomials get higher, resulting in more potential collision-causing roots. This introduces

a trade-off in application design. If one wants to guarantee some maximum collision probability ϵ and the hash-key is chosen from a set of k elements, then the length of messages to be hashed must be limited to around $k\epsilon$ words. The larger the key-set size k used in the hashing polynomial, the more words can be hashed before reaching the allowable collision probability q. But, to make the key-set size significantly larger requires the polynomial to be computed over a larger prime-field, and in general, as the prime p is increased, so is the time needed to evaluate the polynomials in $\mathbb{Z}_p$. As one can see by examining the timing results for PolyQ32 and PolyQ64, the move from a prime close to 2^{32} to one close to 2^{64} increases the number of cycles-per-byte required to hash a message by nearly 50%.[2]

Can we have the best of both worlds: a hash function which is as fast as PolyQ32 but can hash messages as long as PolyQ64, without having intollerably high collision probability? This is the goal which motivates this section. We approach the problem with the belief that most strings being hashed are short, but that a generalized hash function should be able to handle well long messages too.

Our idea is to hash short messages (up to some fixed number of bits ℓ) directly with PolyQ32, but hash messages longer than ℓ bits with a hybrid scheme. Let us say that message M is longer than ℓ bits. To hash M we first partition it into its ℓ bit prefix M_1 and the remainder M_2, so that $M_1 \parallel M_2 = M$. The hash of M under our hybrid scheme is then $\mathsf{PolyQ64}(k_2, \mathsf{PolyQ32}(k_1, M_1) \parallel M_2)$. In this manner, the first ℓ bits of M is hashed with a fast hash function (which cannot safely hash long messages), and if there is any of the string left after hashing its prefix, the remainder is hashed with a slower hash function (which can safely hash longer messages). The parameter ℓ depends on the maximum desirable collision bound and how long a message can be before the fast hash function approaches this bound.

As an example, let us say that we want to hash messages and have a collision bound of no more than 2^{-16}. If we were to hash solely with PolyQ32, then we could hash no messages longer than around 2^{17} bits. Alternatively, we could hash with only PolyQ64 and would then be able to hash strings as long as 2^{39} bits before allowing 2^{-16} collision probability, but at a much slower rate than PolyQ32. Under our scheme, if a message M is shorter than 2^{17} bits, then the hash result is simply $\mathsf{PolyQ32}(M)$; whereas if M is longer than 2^{17} bits, then the hash is calculated as $\mathsf{PolyQ64}(\mathsf{PolyQ32}(M_1) \parallel M_2)$ where M_1 is the 2^{17}-bit prefix of M. Such a construction is fast on short messages, but handles well long messages too. If messages were anticipated to be longer than 2^{39}, then a function PolyQ96, employing a 96-bit prime modulus, could be defined analogously and be employed as a third-stage polynomial. This ramping-up of the prime modulus used in the polynomial evaluations gives the construction its name: *Ramped polynomial hashing.*

One might expect the collision bound of such a hybrid approach to be approximately the *sum* of the collision bounds of each of its constituent functions, but as the following theorem shows, the overall collision bound is instead only the *maximum* of the functions.

The following theorem and proof address PolyR32_64, the ramped polynomial hash of Figure 5. This concrete hash function hashes up to 2^{14} bits (equivalent to 2^9 32-bit words) using the fast PolyQ32 function, and allows a total message length of up to 2^{36} bits (or 2^{30} 64-bit words). In the following theorem and proof, for increased generality, we use parameters ℓ and m instead of the numbers of words 2^9 and 2^{30}.

Theorem 5.1 *Let $\ell = 2^9$ and $m = 2^{30}$. Let $M \neq M'$ be messages no longer than $64m$ bits. Then* $\Pr[k_1 \xleftarrow{R} K_{32}; k_2 \xleftarrow{R} K_{64} : \mathsf{PolyR32\_64}(k_1, k_2, M) = \mathsf{PolyR32\_64}(k_1, k_2, M')] \leq \max(\ell 2^{-28}, m2^{-49}) +$

[2]Some of this difference is an artifact of the fact that the Pentium II natively supports multiplication of 32-bit operands to a 64-bit result, but not the multiplication of 64-bit operands to a 128-bit result. Most processors will display this type of threshold behavior when operands exceed well-supported lengths.

2^{-50}.

Proof. Let M and M' be messages, and imagine partitioning them into $M = M_1 \parallel M_2$ and $M' = M'_1 \parallel M'_2$ so that M_1 and M'_1 are the first 32ℓ bits of M and M'. If M is shorter than 32ℓ bits, then $M_1 = M$ and M_2 is empty. Likewise, if M' is shorter than 32ℓ bits, then $M'_1 = M'$ and M'_2 is empty. We ignore all padding issues in this discussion, assuming that standard padding techniques are used to bring M and M' to appropriate lengths. Let $k_1 \xleftarrow{R} K_{32}$ and $k_2 \xleftarrow{R} K_{64}$ be randomly chosen keys. We define the following values here for convenience, but all probabilities in this proof are assumed to be taken over these random choices of k_1 and k_2.

$$h_1 = \mathsf{num2str}(\mathsf{PolyQ32}(k_1, M_1), 64) \quad \text{and} \quad h'_1 = \mathsf{num2str}(\mathsf{PolyQ32}(k_1, M'_1), 64)$$
$$h_2 = \mathsf{num2str}(\mathsf{PolyQ64}(k_2, h_1 \parallel M_2), 64) \quad \text{and} \quad h'_2 = \mathsf{num2str}(\mathsf{PolyQ64}(k_1, M'_2), 64)$$

Depending on the lengths of M and M', the result of hashing M will be h_1 or h_2 and the result of hashing M' will be h'_1 or h'_2. We examine several cases for the relative lengths of M and M'.

Case 1: Different lengths, same ramp. Here we examine the case where the messages M and M' are different lengths, but are both either longer than 32ℓ bits or both no longer. If both are no longer than 32ℓ bits then a collision occurs if $h_1 = h'_1$. But, $M_1 = M$ and $M'_1 = M'$ differ in length which means (by Proposition 4.3) that $\Pr[h_1 = h'_1] \leq \ell 2^{-28}$. If both M and M' are longer than 32ℓ bits then a collision occurs if $h_2 = h'_2$. But, $h_1 \parallel M_2$ and $h'_1 \parallel M'_2$ also differ in length which means (by Proposition 4.4) that $\Pr[h_2 = h'_2] \leq m2^{-49}$

Case 2: Different lengths, different ramp. If M is longer than 32ℓ bits and M' is not, then a collision occurs only if $h'_1 = h_2$. Expanding the h_2 term, we see that a collision only occurs if $h'_1 = \mathsf{num2str}(\mathsf{PolyQ64}(k_2, h_1 \parallel M_2), 64)$. If we fix k_1 to an arbitrary value, then h_1 and h'_1 become fixed as well, and the probability of collision then depends only on the selection of k_2. The string $h_1 \parallel M_2$ is partitioned by the PolyQ64 algorithm into 64-bit strings and then transformed by DoubleTransform into some sequence $x_0, x_1, \ldots, x_{n \leq 2m}$ of elements from $\mathbb{Z}_{p(64)}$. This sequence is then used in the summation $\sum_{i=0}^{n} x_i k_2^{n-i} \bmod p(64)$ to calculate the final hash result. A collision occurs if the result of this summation is h'_1, or alternatively when $\sum_{i=0}^{n} x_i k_2^{n-i} - h'_1 \bmod p(64) = 0$. The Fundamental Theorem of Algebra applies to this last polynomial, meaning there are no more than $n \leq 2m$ values for k_2 which satisfy it. Thus, $\Pr[h'_1 = h_2] \leq 2m2^{-29} = m2^{-28}$.

Case 3: Equal length messages, last ramp different. If M and M' are equal length, longer than 32ℓ bits and $M_2 \neq M'_2$, then (by Proposition 4.4) $\Pr[h_2 = h'_2] \leq m2^{-49}$ because $h_1 \parallel M_2$ and $h'_1 \parallel M'_2$ are distinct. Similarly, if M and M' are the same length, no longer than 32ℓ bits and $M_1 \neq M'_1$, then (by Proposition 4.3) $\Pr[h_1 = h'_1] \leq m2^{-28}$ because M_1 and M'_1 are distinct.

Case 4: Equal length messages, last ramp same. If M and M' are equal length and longer than 32ℓ bits, and $M_1 \neq M'_1$ but $M_2 = M'_2$, then there are two opportunities for a collision to take place. First, if $\mathsf{PolyQ32}(k_1, M_1) = \mathsf{PolyQ32}(k_1, M'_1)$, then the strings $h_1 \parallel M_2$ and $h'_1 \parallel M'_2$ are equal, guaranteeing that h_2 and h'_2 collide. The probability of this event is no more than $\ell 2^{-28}$. Second, if $h_1 \neq h'_1$, then a collision can still occur if $\mathsf{PolyQ32}(k_2, h_1 \parallel M_1) = \mathsf{PolyQ32}(k_2, h'_2 \parallel M'_2)$. One might think that this is an event with up to $m2^{-49}$ probability, but it is not. Because $M_2 = M'_2$, the strings $h_1 \parallel M_1$ and $h'_1 \parallel M'_1$ only differ in their first 64-bit word. The collision event when hashing such strings takes the form $(h_1 - h'_1)k_2^n = 0$, which can only be satisfied if $k_2 = 0$, a 2^{-50} probability event. Thus, the total probability of collision in this case is bounded by $\ell 2^{-28} + 2^{-50}$. $\diamondsuit$

5.1 Security Notes

If a lower collision probability is desired, one can hash messages multiple times, using a different key for each message hash. A hash function which has an ϵ collision bound when hashing once with a random key, has an ϵ^2 collision bound when hashing twice with two random keys, and an ϵ^3 collision bound when hashed with three keys, etc.

Also, all of the theorems in this work have been stated in terms of collisions (ie. the difference between the result of evaluating the hash of two distinct messages is zero). It is a simple matter to tweak the algorithms and proofs to show that the probability that the difference between the hash of two distinct messages being a particular constant is bounded by the same ϵ. This version of universal hashing ("delta"-universal) is required in some message authentication schemes.

References

[1] Afanassiev, V., Gehrmann, C., and Smeets, B. Fast message authentication using efficient polynomial evaluation. In *Proceedings of the 4th Workshop on Fast Software Encryption* (1997), vol. 1267, Springer-Verlag, pp. 190–204.

[2] Bernstein, D. Floating-point arithmetic and message authentication. Unpublished manuscript, `http://cr.yp.to/papers.html`, 2000.

[3] Black, J., Halevi, S., Krawczyk, H., Krovetz, T., and Rogaway, P. UMAC: Fast and secure message authentication. In *Advances in Cryptology – CRYPTO '99* (1999), vol. 1666 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 216–233.

[4] Black, J., Halevi, S., Krawczyk, H., Krovetz, T., and Rogaway, P. UMAC: Fast and secure message authentication. In *Advances in Cryptology – CRYPTO '99* (1999), vol. 1666 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 216–233.

[5] Bosselaers, A., Govaerts, R., and Vandewalle, J. Fast hashing on the Pentium. In *Advances in Cryptology – CRYPTO '96* (1996), vol. 1109 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 298–312. Updated timing at `http://www.esat.kuleuven.ac.be/ bosselae/fast.html`.

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

[7] Halevi, S., and Krawczyk, H. MMH: Software message authentication in the Gbit/second rates. In *Proceedings of the 4th Workshop on Fast Software Encryption* (1997), vol. 1267, Springer-Verlag, pp. 172–189.

[8] Krawczyk, H. LFSR-based hashing and authentication. In *Advances in Cryptology – CRYPTO '94* (1994), vol. 839 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 129–139.

[9] Rogaway, P. Bucket hashing and its application to fast message authentication. In *Advances in Cryptology – CRYPTO '95* (1995), vol. 963 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 313–328.

[10] Shoup, V. On fast and provably secure message authentication based on universal hashing. In *Advances in Cryptology – CRYPTO '96* (1996), vol. 1109 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 313–328.

[11] Tsudik, G. Message authentication with one-way hash functions. *Computer Communications Review 22* (1992), 29–38.

[12] Wegman, M., and Carter, L. New hash functions and their use in authentication and set equality. *J. of Computer and System Sciences 22* (1981), 265–279.

A Fully Parameterized: PolyQ, PolyR

The body of this paper developed a two-stage ramped polynomial hash function PolyR32_64 using polynomials over 32- and 64-bit prime fields. The concrete choices made for PolyR32_64 were designed especially for a message authentication code. In the MAC, we needed a universal hash function which would guarantee a collision bound of at most 2^{-16} and would typically be applied to messages no longer than a few dozen bytes. But, the hash function must also be able to process huge inputs, too, and still guarantee a bound of at most 2^{-16}. These requirements led to the development of ramped polynomial hashing in general, and in the choice of the 32- and 64-bit prime fields, and associated crossover points, used in the body of this paper.

Other collision bounds and message lengths not addressed by PolyR32_64 are likely, and so we present in this appendix fully parameterized versions of the hashes called PolyQ and PolyR. For each of the algorithms we state their collision bounds as a theorem, but give no proofs. The proofs are straightforward extensions of those given in the body of the paper.

Proposition A.1 *Let v be any positive integer, let $K \subseteq \mathbb{Z}_{p(v)}$ be any subset of points in the field $\mathbb{Z}_{p(v)}$, and let $2^{v-1} \leq d \leq p(v)$. For any positive integers $\ell \leq n$ and distinct messages $M \in \{0,1\}^{lv}$ and $M' \in \{0,1\}^{nv}$,* $\Pr[k \xleftarrow{R} K : \mathsf{PolyQ}[K,v,d](k,M) = \mathsf{PolyQ}[K,v,d](k,M')] \leq 2n/|K|$.

Proposition A.2 *Let all of the parameters from Figure 6 be fixed. For any distinct messages M and M', each shorter than $\sum_{1 \leq i \leq r} \ell_i v_i$ bits,* $\Pr[\mathbf{k} \xleftarrow{R} \mathbf{K} : \mathsf{PolyR}(\mathbf{k},M) = \mathsf{PolyR}(\mathbf{k},M')] \leq$

$$\max_{1 \leq i \leq r} \left\{ \frac{2\ell_i}{|K_i|} \right\} + \sum_{i=2}^{r} \frac{1}{|K_i|} \; .$$

$r \geq 1$	:	Length of $\mathbf{v}$, $\mathbf{l}$, $\mathbf{K}$ vectors used in PolyR.
$\mathbf{v} = (v_1, \ldots, v_r)$	:	Word-lengths used in PolyR, with $1 < v_1 < \cdots < v_r$.
$\mathbf{l} = (\ell_1, \ldots, \ell_r)$	:	Message lengths used in PolyR, with $\ell_i \geq 1$ for $1 \leq i \leq r$.
$\mathbf{d} = (d_1, \ldots, d_r)$	:	Domain bounds used in PolyR, with $2^{v_i - 1} \leq d_i \leq p(v_i)$.
$\mathbf{K} = (K_1, \ldots, K_r)$	:	Key-sets used in PolyR, with $K_i \subseteq \mathbb{Z}_{p(v_i)}$ for $1 \leq i \leq r$.

Figure 6: *Parameters used in the fully parameterized* PolyR *algorithm. Fixing these parameters fixes the algorithm definition specified in Figure 8.*

algorithm $\mathsf{PolyQ}[K, v, d](k, M)$
// *Parameters: "Key set"* $K \subseteq \mathbb{Z}_{p(v)}$*, "word length"* $v \geq 1$ *and "domain-bound"* d*.*
// *Input:* $k \in K$ *and* $M \in (\{0,1\}^v)^+$*.*
// *Output:* $y \in \mathbb{Z}_{p(v)}$*.*
$\mathit{offset} \leftarrow 2^v - p(v)$ // *Constant for translating out-of-range words*
$\mathit{marker} \leftarrow p(v) - 1$ // *Constant for indicating out-of-range words*
$n \leftarrow |M|/v$
$M_1 \parallel \ldots \parallel M_n \leftarrow M$, // *Break* M *into word size chunks*
where $|M_1| = \cdots = |M_n| = v$
$y \leftarrow 1$ // *Set highest coefficient to 1*
for $i \leftarrow 1$ **to** n **do**
$m \leftarrow \mathsf{str2num}(M_i)$
if $(m \geq d)$ **then** // *If word is not in range, then*
$y \leftarrow ky + \mathit{marker}$ **mod** $p(v)$ // *Marker indicates out-of-range*
$y \leftarrow ky + (m - \mathit{offset})$ **mod** $p(v)$ // *Offset* m *back into range*
else
$y \leftarrow ky + m$ **mod** $p(v)$ // *Otherwise hash in-range word*
return y

Figure 7: *The* PolyQ *algorithm, parameterized on key-set* K*, word-length* v *and domain-bound* d*.*

algorithm $\mathsf{PolyR}(\mathbf{k}, M)$
// *Parameters: Uses "vector length"* r*, "word-length vector"* $\mathbf{v}$*,*
// *"message-length vector"* $\mathbf{l}$*, "domain-bounds vector"* $\mathbf{d}$*, and "key-set vector"* $\mathbf{K}$*.*
// *Input:* $\mathbf{k} = (k_1, \ldots, k_r)$ *with* $k_i \in K_i$ *for* $1 \leq i \leq r$ *and* $M \in \{0,1\}^*$*.*
// *Output:* $Y \in \{0,1\}^{v_r}$*.*
$\mathit{prepend} \leftarrow \varepsilon$ // *Initially no string to prepend*
$i \leftarrow 1$ // *Index for* $\mathbf{v}$, $\mathbf{l}$, $\mathbf{K}$ *vectors*
while $(|M| > \ell_i v_i)$ **do** // *While more than one ramp-level remains*
if $(i = r)$ **then return** Error // *Message too long*
$T \leftarrow M[1 \ldots \ell_i v_i]$ // *Extract the string to be hashed under* $p(v_i)$
$M \leftarrow M[\ell_i v_i + 1 \ldots |M|]$
$y \leftarrow \mathsf{PolyQ}[K_i, v_i, d_i](k_i, \mathit{prepend} \parallel T)$ // *Hash in* $\mathbb{Z}_{p(v_i)}$*, prepending previous hash*
$\mathit{prepend} \leftarrow \mathsf{num2str}(y, v_{i+1})$ // *Update* prepend *for next ramp-level*
$i \leftarrow i + 1$
$M \leftarrow \mathsf{padonezero}(M, v_i)$ // *Final ramp-level needs bijective padding*
$y \leftarrow \mathsf{PolyQ}[K_i, v_i, d_i](k_i, \mathit{prepend} \parallel M)$ // *Hash in* $\mathbb{Z}_{p(v_i)}$*, prepending previous hash*
$Y \leftarrow \mathsf{num2str}(y, v_r)$ // *Convert to string*
return Y

Figure 8: *The* PolyR *algorithm. Parameters are described in Figure 6.*