Evaluating Bernstein-Rabin-Winograd Polynomials

Debrup Chakraborty Sebati Ghosh Palash Sarkar
Indian Statistical Institute
203, B.T.Road, Kolkata, India - 700108.
{debrup,sebati_r,palash}@isical.ac.in

April 13, 2017

Abstract

We describe a non-recursive algorithm which can efficiently evaluate Bernstein-Rabin-Winograd polynomials with variable number of blocks.
Keywords: universal hash function, BRW polynomials.

1 Introduction

In [1], Bernstein built upon a previous work due to Rabin and Winograd [4] to propose a family of polynomials which have been called the BRW polynomials in [5]. A BRW polynomial is constructed from $m \geq 0$ field elements. For $m \geq 3$, a BRW polynomial constructed from m field elements can be evaluated using $\lfloor m/2 \rfloor$ field multiplications and $\lfloor \lg m \rfloor$ squarings. The importance of such polynomials for constructing hash functions with low collision and differential probabilities has been discussed in [1]. Hardware implementation of BRW polynomials has been reported in [3] and a recent work [2] reports the software implementation of BRW polynomials for $m = 31$.

The definition of BRW polynomials is recursive. This makes it difficult to have a software implementation of BRW polynomials where m can vary. To the best of our knowledge, no prior work has reported any algorithm for evaluating BRW polynomials with variable m. In this work, we describe an efficient algorithm for this task.

2 BRW Polynomials

Let $\mathbb{F}$ be a finite field. For $m \geq 0$, $\mathsf{BRW}_\tau(M_1, M_2, \cdots, M_m)$ with $M_1, \ldots, M_m \in \mathbb{F}$ is a polynomial in the variable τ and is defined as follows:

- $\mathsf{BRW}_\tau() = 0$;
- $\mathsf{BRW}_\tau(M_1) = M_1$;
- $\mathsf{BRW}_\tau(M_1, M_2) = M_1\tau + M_2$;
- $\mathsf{BRW}_\tau(M_1, M_2, M_3) = (\tau + M_1)(\tau^2 + M_2) + M_3$;
- $\mathsf{BRW}_\tau(M_1, M_2, \cdots, M_m)$
  $= \mathsf{BRW}_\tau(M_1, \cdots, M_{t-1})(\tau^t + M_t) + \mathsf{BRW}_\tau(M_{t+1}, \cdots, M_m)$;
  if $t \in \{4, 8, 16, 32, \cdots\}$ and $t \leq m < 2t$.

3 Algorithm

The following data structures and variables are used in the algorithm.

isDef$[0, \ldots]$: a bit array;
res$[0, \ldots]$: an array where partial results are stored;
keyPow$[0, \ldots]$: the j-th location stores τ^{2^j};
ℓ: current length of both isDef and res.

The interpretation of the two arrays is as follows: for $1 \le j \le \ell$, isDef$[j] = 1$ if and only if res$[j]$ holds a valid partial result. When i blocks (field elements) have been processed, the value of ℓ is $\lfloor \lg i \rfloor$.

The following external functions are used.

polyMult(A, B): returns the product of the polynomials A and B without reduction;
reduce(A): reduces the polynomial A;
getBlks(k): returns $(M_1, \ldots, M_k, t)$;
EOF: returns true if there are no more blocks left and false otherwise.

For the output $(M_1, \ldots, M_k, t)$ returned by getBlks(k), $1 \le t \le k$ and blocks $M_1, \ldots, M_t$ are the next t blocks from the buffer; if $t < k$, then $M_{t+1}, \ldots, M_k$ are not defined.

The algorithm for computing variable length BRW polynomials is the following. The algorithm assumes that there is at least one block.

Algorithm $\mathcal{A}(\tau, M_1, M_2, \ldots)$

```
i ← 1; ℓ ← 1; keyPow[0] = τ; keyPow[1] = τ^2;
while not EOF do
    (M_i, M_{i+1}, M_{i+2}, M_{i+3}, t) ← getBlks(4);
    if t = 4 then
        res[0] ← polyMult(M_i + keyPow[0], M_{i+1} + keyPow[1]) + M_{i+2};
        j ← 1; tmp ← res[0];
        while (j < ℓ and isDef[j] = 1) do tmp ← tmp + res[j]; j ← j + 1; end do;
        if j = ℓ then ℓ ← ℓ + 1; keyPow[ℓ] ← keyPow[ℓ − 1]^2; end if;
        res[j] ← polyMult(reduce(tmp), M_{i+3} + keyPow[j + 1]);
        isDef[j] ← 1;
        for k = 0 to j − 1 do isDef[k] ← 0; end do;
    else
        if t = 1 then res[0] ← M_i;
        if t = 2 then res[0] ← polyMult(M_i, keyPow[0]) + M_{i+1};
        if t = 3 then res[0] ← polyMult(M_i + keyPow[0], M_{i+1} + keyPow[1]) + M_{i+2};
        isDef[0] ← 1;
    end if;
    i ← i + t;
end do;
tmp ← 0;
for j = 0 to ℓ − 1 do
    if isDef[j] = 1 then tmp ← tmp + res[j]; end if;
end do;
return reduce(tmp).
```

The array isDef can be implemented using a b-bit unsigned integer: the value of the j-th can be obtained as $(\mathsf{isDef} \gg j)$ `and` $1)$ (required in Steps 7 and 22); the value of the j-th bit can be set to one using $\mathsf{isDef} \leftarrow (\mathsf{isDef}$ `or` $(1 \ll j))$ (required in Steps 10 and 16); the j least significant bits of isDef can be set to 0 using $\mathsf{isDef} \leftarrow (\mathsf{isDef}$ `and` $(1^b \ll j))$ (required in Step 11).

4 Modification of the Algorithm: Number of Blocks is Known

If the number of blocks m is known, then Algorithm $\mathcal{A}$ can be simplified to improve the efficiency. For this algorithm, we assume that $\mathsf{getBlks}(4)$ returns exactly 4 blocks.

Algorithm $\mathcal{B}(\tau, M_1, \ldots, M_m)$, $m \geq 1$
1. $\mathsf{keyPow}[0] = \tau$;
2. if $m > 2$ then
3. for $j = 1$ to $\lfloor \lg m \rfloor$ do $\mathsf{keyPow}[j] = \mathsf{keyPow}[j-1]^2$; end do;
4. end if;
5. $\mathsf{isDef}[0] = 0$;
6. if $m \geq 4$ then
7. for $j = 1$ to $\lfloor \lg m \rfloor - 1$ do $\mathsf{isDef}[j] = 0$; end do;
8. end if;
9. for $i = 1$ to $\lfloor m/4 \rfloor$ do
10. $(M_{4i-3}, M_{4i-2}, M_{4i-1}, M_{4i}) \leftarrow \mathsf{getBlks}(4)$;
11. $\mathsf{res}[0] \leftarrow \mathsf{polyMult}(M_{4i-3} + \mathsf{keyPow}[0], M_{4i-2} + \mathsf{keyPow}[1]) + M_{4i-1}$;
12. $j \leftarrow 1$; $\mathsf{tmp} \leftarrow \mathsf{res}[0]$;
13. while $(\mathsf{isDef}[j] = 1)$ do $\mathsf{tmp} \leftarrow \mathsf{tmp} + \mathsf{res}[j]$; $j \leftarrow j + 1$; end do;
14. $\mathsf{res}[j] \leftarrow \mathsf{polyMult}(\mathsf{reduce}(\mathsf{tmp}), M_{4i} + \mathsf{keyPow}[j+1])$;
15. $\mathsf{isDef}[j] \leftarrow 1$;
16. for $k = 0$ to $j - 1$ do $\mathsf{isDef}[k] \leftarrow 0$; end do;
17. end do;
18. if $m \bmod 4 = 1$ then $\mathsf{res}[0] \leftarrow M_m$; end if;
19. if $m \bmod 4 = 2$ then $\mathsf{res}[0] \leftarrow \mathsf{polyMult}(M_{m-1}, \mathsf{keyPow}[0]) + M_m$; end if;
20. if $m \bmod 4 = 3$ then $\mathsf{res}[0] \leftarrow \mathsf{polyMult}(M_{m-2} + \mathsf{keyPow}[0], M_{m-1} + \mathsf{keyPow}[1]) + M_m$; end if;
21. if $m \bmod 4 \neq 0$ then $\mathsf{isDef}[0] \leftarrow 1$; end if;
22. $\mathsf{tmp} \leftarrow 0$;
23. for $j = 0$ to $\lfloor \lg m \rfloor - 1$ do
24. if $\mathsf{isDef}[j] = 1$ then $\mathsf{tmp} \leftarrow \mathsf{tmp} + \mathsf{res}[j]$; end if;
25. end do;
26. return $\mathsf{reduce}(\mathsf{tmp})$.

5 Modification of the Algorithm: Loop Unrolling

In Algorithm $\mathcal{B}$, the main loop first computes $\mathsf{polyMult}(M_{4i-3} + \mathsf{keyPow}[0], M_{4i-2} + \mathsf{keyPow}[1]) + M_{4i-1}$ and then merges it with an appropriate segment of previously computed partial result. Note that $\mathsf{polyMult}(M_{4i-3} + \mathsf{keyPow}[0], M_{4i-2} + \mathsf{keyPow}[1]) + M_{4i-1}$ is essentially the computation of $\mathsf{BRW}_\tau(M_{4i-3}, M_{4i-2}, M_{4i-1})$ with the only difference that the final result is not reduced.

Let $t \geq 2$ and suppose that the main loop processes 2^t blocks at a time in the following manner. First $\mathsf{BRW}_\tau(M_{2^t \cdot i-(2^t-1)}, M_{2^t \cdot i-(2^t-2)}, \ldots, M_{2^t \cdot i-1})$ is computed without reducing the final result.

Next, this is appropriately merged with previously computed partial results. In Algorithm $\mathcal{B}$ we have $t = 2$. Allowing t to be greater than 2 essentially means an unrolling of the loop. To be able to do this, we introduce a modification of BRW where the final reduction is not applied.

- $\mathsf{unreducedBRW}_\tau() = 0$;
- $\mathsf{unreducedBRW}_\tau(M_1) = M_1$;
- $\mathsf{unreducedBRW}_\tau(M_1, M_2) = \mathsf{polyMult}(M_1, \tau) + M_2$;
- $\mathsf{unreducedBRW}_\tau(M_1, M_2, M_3) = \mathsf{polyMult}((\tau + M_1), (\tau^2 + M_2)) + M_3$;
- $\mathsf{unreducedBRW}_\tau(M_1, M_2, \cdots, M_m)$
  $= \mathsf{polyMult}(\mathsf{BRW}_\tau(M_1, \cdots, M_{t-1}), (\tau^t + M_t)) + \mathsf{unreducedBRW}_\tau(M_{t+1}, \cdots, M_m)$;
  if $t \in \{4, 8, 16, 32, \cdots\}$ and $t \leq m < 2t$.

The modified algorithm with loop unrolling can now be described as follows.

Algorithm $\mathcal{C}(\tau, M_1, \ldots, M_m, t)$, $m \geq 1$, $t \geq 2$

$\mathsf{keyPow}[0] = \tau$;
if $m > 2$ then
    for $j = 1$ to $\lfloor \lg m \rfloor$ do $\mathsf{keyPow}[j] = \mathsf{keyPow}[j-1]^2$; end do;
end if;
$\mathsf{isDef}[0] = 0$;
if $m \geq 2^t$ then
    for $j = 1$ to $\lfloor \lg m \rfloor - t + 1$ do $\mathsf{isDef}[j] = 0$; end do;
end if;
for $i = 1$ to $\lfloor m/2^t \rfloor$ do
    $(M_{2^t \cdot i - (2^t - 1)}, \ldots, M_{2^t \cdot i}) \leftarrow \mathsf{getBlks}(2^t)$;
    $\mathsf{res}[0] \leftarrow \mathsf{unreducedBRW}_\tau(M_{2^t \cdot i - (2^t - 1)}, \ldots, M_{2^t \cdot i - 1})$;
    $j \leftarrow 1$; $\mathsf{tmp} \leftarrow \mathsf{res}[0]$;
    while $(\mathsf{isDef}[j] = 1)$ do $\mathsf{tmp} \leftarrow \mathsf{tmp} + \mathsf{res}[j]$; $j \leftarrow j + 1$; end do;
    $\mathsf{res}[j] \leftarrow \mathsf{polyMult}(\mathsf{reduce}(\mathsf{tmp}), M_{2^t \cdot i} + \mathsf{keyPow}[j + t - 1])$;
    $\mathsf{isDef}[j] \leftarrow 1$;
    for $k = 0$ to $j - 1$ do $\mathsf{isDef}[k] \leftarrow 0$; end do;
end do;
$r = m \bmod 2^t$;
if $r > 0$ then $\mathsf{tmp} \leftarrow \mathsf{unreducedBRW}_\tau(M_{m-r+1}, \ldots, M_m)$;
else $\mathsf{tmp} \leftarrow 0$;
end if;
for $j = 1$ to $\lfloor \lg m \rfloor - t + 1$ do
    if $\mathsf{isDef}[j] = 1$ then $\mathsf{tmp} \leftarrow \mathsf{tmp} + \mathsf{res}[j]$; end if;
end do;
return $\mathsf{reduce}(\mathsf{tmp})$.

6 Timing Results

We have implemented Algorithm-$\mathcal{C}$ in Intel intrinsics for $n = 128$ and $t = 2, 3, 4$ and 5. The corresponding timing results that were obtained are shown in Tables 1 and 2. The column headers provide the message size in bytes and the entries in the tables are in cycles per byte.

Table 1: Indicative timing results on Haswell. The basic field multiplications were implemented using Karatsuba.

	512	1024	4096	8192
$t = 2$	0.94	0.77	0.60	0.57
$t = 3$	1.01	0.84	0.68	0.65
$t = 4$	0.92	0.75	0.58	0.55
$t = 5$	0.86	0.68	0.51	0.48

Table 2: Indicative timing results on Skylake. The basic field multiplications were implemented using schoolbook.

	512	1024	4096	8192
$t = 2$	0.72	0.57	0.43	0.40
$t = 3$	0.82	0.68	0.54	0.51
$t = 4$	0.71	0.57	0.44	0.41
$t = 5$	0.68	0.52	0.38	0.34

References

[1] Daniel J. Bernstein. Polynomial evaluation and message authentication, 2007. `http://cr.yp.to/papers.html#pema`.

[2] Debrup Chakraborty, Sebati Ghosh, and Palash Sarkar. A fast single-key two-level universal hash function. *IACR Trans. Symmetric Cryptol.*, 2017(1):106–128, 2017.

[3] Debrup Chakraborty, Cuauhtemoc Mancillas-López, Francisco Rodríguez-Henríquez, and Palash Sarkar. Efficient hardware implementations of BRW polynomials and tweakable enciphering schemes. *IEEE Trans. Computers*, 62(2):279–294, 2013.

[4] Michael O. Rabin and Shmuel Winograd. Fast evaluation of polynomials by rational preparation. *Communications on Pure and Applied Mathematics*, 25:433–458, 1972.

[5] Palash Sarkar. Efficient tweakable enciphering schemes from (block-wise) universal hash functions. *IEEE Trans. Information Theory*, 55(10):4749–4760, 2009.