A Fast Single-Key Two-Level Universal Hash Function

Debrup Chakraborty   Sebati Ghosh   Palash Sarkar

Indian Statistical Institute, Kolkata

7th March, 2017
Outline

1. Introduction
2. Our Contribution
3. Implementation Results
4. Other Contributions
Outline

1. Introduction
2. Our Contribution
3. Implementation Results
4. Other Contributions
Universal Hash Function

- Was introduced by Carter and Wegman in 1979.
- It is an important primitive in cryptography.
- Two main objectives:
  - Reducing the computation time (specially multiplication count)
  - Reducing the key size
<table>
<thead>
<tr>
<th>scheme</th>
<th># mult</th>
<th># sqr</th>
<th>key size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horner</td>
<td>(\ell - 1)</td>
<td>–</td>
<td>single field element</td>
</tr>
<tr>
<td>Bernstein-Rabin-Winograd (BRW)</td>
<td>(\lfloor \ell/2 \rfloor)</td>
<td>(\lfloor \lg \ell \rfloor)</td>
<td>single field element</td>
</tr>
</tbody>
</table>

**Table**: Univariate polynomial based hashing for message consisting of \(\ell\) blocks for \(\ell \geq 3\).
BRW polynomials based hash function is advantageous over Horner in terms of operation (field mult.) count.

Problem is BRW polynomials are inherently recursive; significant implementation overhead for variable length messages.

If applied on fixed length messages, this difficulty disappear and we can get the benefit of speed.

Horner can handle arbitrary length messages quite easily.
Objective

- Two-level Hash Function: to combine BRW and Horner to enjoy the benefits of both; apply BRW on fixed length components of the input message and combine the outputs using Horner.

- Use a single field element as the key.

- Propose two-level hash for handling a single binary string (Hash2L) and a vector of binary strings (vecHash2L).

- Optimised implementations of Hash2L over the fields \( \mathbb{F}_2^{128} \) and \( \mathbb{F}_2^{256} \).
Outline

1. Introduction
2. Our Contribution
3. Implementation Results
4. Other Contributions
Outline

1. Introduction

2. Our Contribution
   - Design
   - Implementation

3. Implementation Results

4. Other Contributions
Hash2L: flowchart
Hash2L: flowchart

- $M$ is input to $\text{pad}_n(\cdot)$
- $\text{pad}_n(M)$ results in division into superblocks
Hash2L: flowchart

- $M$
- $\text{pad}_n(\cdot)$
- $\text{pad}_n(M)$
- $\tau$
- $\text{division into superblocks}$
- $M_1$
- $M_2$
- $\ldots$
- $M_\ell$
- $\text{BRW}_\tau(\cdot)$
Hash2L: flowchart

Introduction
Our Contribution
Implementation Results
Other Contributions

Design
Implementation
Hash2L: flowchart

```
M
  | pad_n(·)
  | pad_n(M)
  |
  | division into superblocks

M_1  M_2  ...  M_ℓ
  | BRW_τ(·)  BRW_τ(·)  ...  BRW_τ(·)
  | BRW_τ(M_1) BRW_τ(M_2) ... BRW_τ(M_ℓ)

τ^{d(η)+1}
  | Horner_τ^{d(η)+1}(·)
  | bin_n(len(M))

τ^2

τ
```
Hash2L: flowchart

\[
\begin{align*}
\text{pad}_n(\cdot) & \quad \text{division into superblocks} \\
\text{BRW}_\tau(\cdot) & \quad \text{BRW}_\tau(\cdot) \\
\text{BRW}_\tau(M_1) & \quad \text{BRW}_\tau(M_2) \\
\text{Horner}_{\tau^{d(\eta)+1}}(\cdot) & \\
\text{Hash2L}_\tau(M) & \quad \text{bin}_n(len(M))
\end{align*}
\]
Hash2L: security

- The AXU-bound for Hash2L is $\frac{\ell(d(\eta)+1)+1}{2^n}$ for two distinct messages $M$ and $M'$ with $\text{len}(M) \geq \text{len}(M')$ and $\ell$ is the number of super-blocks in $M$. Here, $\eta$ is the number of blocks in a full super-block.

Note: The last super-block may be a partial one.
Outline

1. Introduction
2. Our Contribution
   - Design
   - Implementation
3. Implementation Results
4. Other Contributions
The implementation uses Intel intrinsics, specially the instruction `pclmulqdq`: takes as input two degree 64 polynomials over $\mathbb{F}_2$ and returns their product as degree 128 polynomial.

Timing measurements on both Haswell and Skylake.
Some major optimisations:

- Batch size: grouping `pclmulqdq` instructions for $m$ independent multiplications together for better instruction pipelining; we have checked for batch sizes $\leq 4$. Finally, we used batch size 3 for $n = 128$ and 1 for $n = 256$ for both BRW and Horner.
Using delayed reduction strategy for computing BRW Polynomials: for \( \eta = 31 \), 8 reductions suffice.
Using delayed reduction strategy for computing BRW Polynomials: for $\eta = 31$, 8 reductions suffice.

\[
\text{BRW}_\tau(m_1, \ldots, m_{31}) = \text{BRW}_\tau(m_1, \ldots, m_{15})(\tau^{16} + m_{16}) + \text{BRW}_\tau(m_{17}, \ldots, m_{31})
\]
Implementation (contd.)

- Using *delayed reduction strategy* for computing BRW Polynomials: for $\eta = 31$, 8 reductions suffice.

$$\text{BRW}_\tau(m_1, \ldots, m_{31}) = \text{BRW}_\tau(m_1, \ldots, m_{15})(\tau^{16} + m_{16}) + \text{BRW}_\tau(m_{17}, \ldots, m_{31})$$

normal strategy:
Using *delayed reduction strategy* for computing BRW Polynomials: for $\eta = 31$, 8 reductions suffice.

\[
\text{BRW}_\tau(m_1, \ldots, m_{31}) = \text{BRW}_\tau(m_1, \ldots, m_{15})(\tau^{16} + m_{16}) + \text{BRW}_\tau(m_{17}, \ldots, m_{31})
\]

normal strategy:

field multiplication; one reduction
Using *delayed reduction strategy* for computing BRW Polynomials: for $\eta = 31$, 8 reductions suffice.

\[
\text{BRW}_\tau(m_1, \ldots, m_{31}) = \text{BRW}_\tau(m_1, \ldots, m_{15})(\tau^{16} + m_{16}) + \text{BRW}_\tau(m_{17}, \ldots, m_{31})
\]

normal strategy:

- field multiplication;
- *one* reduction

*one* final reduction
Implementation (contd.)

- Using *delayed reduction strategy* for computing BRW Polynomials: for $\eta = 31$, 8 reductions suffice.

$$BRW_{\tau}(m_1, \ldots, m_{31})$$

$$= \quad BRW_{\tau}(m_1, \ldots, m_{15})(\tau^{16} + m_{16}) \quad + \quad BRW_{\tau}(m_{17}, \ldots, m_{31})$$

Normal strategy:

- Field multiplication; *one* reduction
- XOR the results
- *One* final reduction
Using *delayed reduction strategy* for computing BRW Polynomials: for $\eta = 31$, 8 reductions suffice.

$$BRW_\tau(m_1, \ldots, m_{31}) = BRW_\tau(m_1, \ldots, m_{15})(\tau^{16} + m_{16}) + BRW_\tau(m_{17}, \ldots, m_{31})$$
Using delayed reduction strategy for computing BRW Polynomials: for $\eta = 31$, 8 reductions suffice.

$$\text{BRW}_\tau(m_1, \ldots, m_{31})$$

$$\quad = \text{BRW}_\tau(m_1, \ldots, m_{15})(\tau^{16} + m_{16}) + \text{BRW}_\tau(m_{17}, \ldots, m_{31})$$

Delayed reduction strategy:
Implementation (contd.)

- Using *delayed reduction strategy* for computing BRW Polynomials: for $\eta = 31$, 8 reductions suffice.

$$BRW_\tau(m_1, \ldots, m_{31}) = BRW_\tau(m_1, \ldots, m_{15})(\tau^{16} + m_{16}) + BRW_\tau(m_{17}, \ldots, m_{31})$$

*Delayed reduction strategy:*

- only polynomial multiplication;
- no reduction
Using *delayed reduction strategy* for computing BRW Polynomials: for $\eta = 31$, 8 reductions suffice.

\[
\text{BRW}_\tau(m_1, \ldots, m_{31}) = \text{BRW}_\tau(m_1, \ldots, m_{15})(\tau^{16} + m_{16}) + \text{BRW}_\tau(m_{17}, \ldots, m_{31})
\]

*Delayed reduction strategy:*  
- only polynomial multiplication;  
- avoid final reduction;  
- no reduction.
Using *delayed reduction strategy* for computing BRW Polynomials: for $\eta = 31$, 8 reductions suffice.

\[
\text{BRW}_\tau(m_1, \ldots, m_{31}) = \text{BRW}_\tau(m_1, \ldots, m_{15}) (\tau^{16} + m_{16}) + \text{BRW}_\tau(m_{17}, \ldots, m_{31})
\]

**Delayed reduction strategy:**
- only polynomial multiplication;
- no reduction;
- avoid final reduction;
- XOR the results and do one reduction on the sum.
Outline

1. Introduction
2. Our Contribution
3. Implementation Results
4. Other Contributions
### Timing Measurements: for $\mathbb{F}_{2^{128}}$

<table>
<thead>
<tr>
<th>length of message in bytes</th>
<th>512</th>
<th>1024</th>
<th>4096</th>
<th>8192</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash2L</td>
<td>0.88</td>
<td>0.687</td>
<td>0.498</td>
<td>0.463</td>
</tr>
<tr>
<td>GHASH (Gueron)</td>
<td>1.15</td>
<td>1.02</td>
<td>0.93</td>
<td>0.91</td>
</tr>
<tr>
<td>POLYVAL (Gueron)</td>
<td>1.09</td>
<td>0.81</td>
<td>0.602</td>
<td>0.567</td>
</tr>
</tbody>
</table>

**Table**: Cycles per byte for computing Hash2L, GHASH and POLYVAL on Haswell.

<table>
<thead>
<tr>
<th>length of message in bytes</th>
<th>512</th>
<th>1024</th>
<th>4096</th>
<th>8192</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash2L</td>
<td>0.667</td>
<td>0.468</td>
<td>0.33</td>
<td>0.301</td>
</tr>
<tr>
<td>GHASH (Gueron)</td>
<td>0.89</td>
<td>0.77</td>
<td>0.67</td>
<td>0.65</td>
</tr>
<tr>
<td>POLYVAL (Gueron)</td>
<td>0.79</td>
<td>0.55</td>
<td>0.369</td>
<td>0.339</td>
</tr>
</tbody>
</table>

**Table**: Cycles per byte for computing Hash2L, GHASH and POLYVAL on Skylake.
### Timing Measurements: for $\mathbb{F}_{2^{256}}$

<table>
<thead>
<tr>
<th>length of message in bytes</th>
<th>512</th>
<th>1024</th>
<th>4096</th>
<th>8192</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash2L</td>
<td>1.4</td>
<td>0.95</td>
<td>0.718</td>
<td>0.67</td>
</tr>
</tbody>
</table>

**Table**: Cycles per byte for computing Hash2L on Haswell.

<table>
<thead>
<tr>
<th>length of message in bytes</th>
<th>512</th>
<th>1024</th>
<th>4096</th>
<th>8192</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash2L</td>
<td>1.11</td>
<td>0.758</td>
<td>0.562</td>
<td>0.525</td>
</tr>
</tbody>
</table>

**Table**: Cycles per byte for computing Hash2L on Skylake.
Another measure

According to bit operations per bit of the digest

- Bernstein and Chou (SAC-2014) report this count for a pseudo-dot product based hash function implementation over $\mathbb{F}_{2^{256}}$, based on the Fast Fourier Transform (FFT) based multiplication algorithm to be 29.

- But, this figure excludes the cost for generating the long key, which is expected to be significant in a platform not supporting AES-NI instructions.

- For Hash2L, this cost is at most about 46 for $\eta = 31$.

- But, in this case there is no hidden cost for generating the key.
Outline

1. Introduction
2. Our Contribution
3. Implementation Results
4. Other Contributions
Appendix

*In the paper you can find the following also:*

- detailed construction of vecHash2L.
- detailed security proofs for both Hash2L and vecHash2L.
- detail on implementation of field multiplication
- precise counts of arithmetic operations for computing BRW.
- more detail on implementation of BRW.
- analysis of timing measurements obtained.
- detail calculation of bit operations count w.r.t. the SAC-2014 paper of Bernstein and Chou.
Thank You!