Using Proposed Vector and Crypto Extensions For Fast and Secure Boot
Several case studies in the use of the proposed cryptographic ISA extensions

G. Richard Newell from Microsemi Corp. and the RISC-V Foundation Security Working Group
With Derek Atkins, Drake Smith, and Michael Caiafa from SecureRF Corp.
For the 2nd day of the 7th RISC-V Workshop: Nov. 29, 2017
Sponsored by Western Digital in Milpitas, CA
All crypto [using the extensions] will be done using the vector registers:

- Public key algorithms will use SW with existing and new vector extension commands
  - Data widths will accommodate all the popular asymmetric algorithms and protocols (very big words!)
  - These work with existing vector instructions `vmul, vadd`; plus instructions are added for use with the Galois Field types that also include an extra register for the field-reduction prime or polynomial (`vmulr, vaddr`)
- Symmetric algorithms and digests will use a new vector opcode `vcrypt`
  - The algorithm and function are encoded into the vector instruction: `vcrypt.aes.enc vcipher, vplain, vkey`
- Vector commands work on one element or multiple elements ($1 \leq vl \leq mvl$) per instruction issue
  - The implementation details: size/area tradeoff, single/multiple lanes, side-channel countermeasures, etc.
    are left up to the implementer
- If strip-mined properly, object code will be portable to all RISC-V implementations with the same feature set, whether small/slow or large/fast
- Profiles will be used to help direct implementations to standard feature sets
- Still a work in progress… subject to change

## Summary of Proposed RISC-V Cryptographic Extensions

- (up to) 32 Vector Registers

### Registers
- Shared Meta Data (CSRs)
- Length specifier(s), Element Width, Data Type, etc.
- ~2 bytes/`reg`
- Width Escapes
- `8 x 8b`
- `3 x 16b`

### Algorithm Assignments
- Vector per-`Meta`
- `Meta`
- `Data`
- `mvl`-1

© 2014 Microsemi Corporation. COMPANY PROPRIETARY
Galois Counter Mode Using the Proposed Crypto Extensions

Initialization Vector (IV)

Encryption Algorithm (e.g. AES)

Keystream

Encrypted-and-Authenticated Data

Multiplication over $\mathbb{GF}_2^{128}$

Authenticated-only Data

$vadd\;vctr,\;vctr,\;vl$
Each time through strip-mine loop, add vl to all counter values

$vcrypt.aes.enc\;vks,\;vctr,\;vkey$
Encrypt counters to get keystream

$vl\;vpt,\;a1$

$vxor\;vct,\;vks,\;vpt$

$vst\;vct,\;a2$
XOR keystream with plaintext to get ciphertext (Increment a1 and a2)

This looks like it has serial dependencies, but it can be vectorized because GF multiplication is associative (See next slide)

Repeat until entire message is consumed


© 2014 Microsemi Corporation. COMPANY PROPRIETARY
Vectorizing GHASH

- **GHASH (iterative description)**
  \[ X_{i+1} = (X_i \oplus M_i) \cdot H \]
  \[ = \ldots (((M_0 \cdot H \oplus M_1) \cdot H \oplus M_2) \cdot H \oplus M_3) \cdot H \ldots ) \cdot H \]

- **GHASH (4-element vector description)**
  \[ = (((M_0 \cdot H^4 \oplus M_4) \cdot H^4 \oplus M_8) \cdot H^4 \oplus \ldots) \cdot H^4 \]
  \[ \oplus (((M_1 \cdot H^4 \oplus M_5) \cdot H^4 \oplus M_9) \cdot H^4 \oplus \ldots) \cdot H^3 \]
  \[ \oplus (((M_2 \cdot H^4 \oplus M_6) \cdot H^4 \oplus M_{10}) \cdot H^4 \oplus \ldots) \cdot H^2 \]
  \[ \oplus (((M_3 \cdot H^4 \oplus M_7) \cdot H^4 \oplus M_{11}) \cdot H^4 \oplus \ldots) \cdot H \]

- Vectorized iterations (2 of 4 shown):
  \[ X_{0,k+1} = (X_{0,k} \oplus M_{4 \cdot k}) \cdot H^4 \]  
  \[ X_{1,k+1} = (X_{1,k} \oplus M_{4 \cdot k+1}) \cdot H^4 \] 

Where “\( \cdot \)” is multiplication in \( \text{GF}_{2^{128}} \) with the reduction polynomial:
\[ g = x^{128} + x^7 + x^2 + x + 1 \]
(preload a register with binary of this)
\( \text{vmulr} \) does this operation if the data type is set to ‘GP128’

\( \oplus \) is addition in \( \text{GF}_{2^{128}} \) and is the same as XOR for any “g”

- Easily extendable to more than 4 elements

Each of the four elements is iterative (like in the equation above), but acting on every fourth message element independently of the other elements.

- Precompute H, H^2, H^3, and H^4
- Use H^4 during the vectorized state iterations
- Post-multiply the four final state elements by H, H^2, H^3, and H^4, respectively, and add the sub-totals to get the final result

Update ciphertext \( vct \) (see previous page)
\( \text{vadd vtemp, vstate, vct} \)
\( \text{vmulr vstate, vtemp, vh4, vredpoly} \)
(perform post-processing after last iteration)
AES-GCM Summary

- AES-GCM performance estimates and comparisons
  - Highly implementation dependent – the RISC-V estimates are rough-order-of-magnitude only
  - RISC-V RV32IVY\(^1\) Assumptions:
    - 16 S-box AES vector functional unit (VFU), not pipelined. 16 clock cycles per 14-round (128-bit) encryption
    - Only one lane of wide-arithmetic w/ a (for example) 4-stage 64x64 pipelined core. VFU does four \(GF_{2^{128}}\) multiplications (incl. reductions) every 20 clock cycles
    - Sixteen 128-bit (16B) elements per vector register (2048b=256B). Message size is 8Kb=1Kbytes=32 blks
    - Load/Store, AES, and \(GF_{2^{128}}\) VFUs can operate in parallel

<table>
<thead>
<tr>
<th>RISC-V RV32IVY</th>
<th>ARM Cortex™-M3(^3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2K clock cycles</td>
<td>192K clock cycles</td>
</tr>
</tbody>
</table>

---

1. Using letter “Y” for the RISC-V Cryptographic Extensions (not approved yet)
SHA-256 Algorithm Summary

8 x 32-bits = 256 bit state $H_{i+1}^t$
- Initialized with a constant $H^0$ for the first round of first compression;
- with previous result for each subsequent round and compression

- 64 Rounds per compression:
  - # of compressions (N) depends on size of message

- Final result (after N x 64 rounds)

- 2048-bits, consumed 32-bits/round
- 2048-bit constant, consumed 32-bits/round

- XOR of 8 terms from earlier message schedule outputs, some resulting from var. shifts and rotates

- 8 x 32-bits = 256 bits $H_i^t$
- State is fed back each round
- At end of 64 rounds (one compression) final state initializes next compression
- After entire message (incl. pad) is compressed, the state is the SHA-256 result

Diagram credit: Wikipedia/Kockmeyer, w/ modifications
SHA-256 Vector Assembly Code Example

```
# SHA-256 using vector/crypto extensions: 4 x 512b of msg compressed per opcode
vsetcfgd  SHA_CFG  # Config 3 vectors: 2reg w/ 4 elements x 64B (512b) bitstrings
                 # and 1reg /w 4elements x 32B (256b) bitstrings
vcrypt.init,SHA256  v3  # Select SHA-256 & initialize fixed constants
vsetvl  t1, a1  # Set veclen temporarily to one element (a1=1)
vcld  v1, a3  # Load 512-bit that includes tail of msg
vcrypt.pad.SHA256  v2, v1, t2  # Pad v1->v1,v2 starting @ bit t2 (SHA256 pad)
vst  v1, a3  # Store first padded 512-bit element
bltu  t2, t3, stripmine  # If t2 < t3 (t3=512), v2 not needed for pad
       t2, t3, stripmine  # If t2 < t3 (t3=512), v2 not needed for pad
       a3, a3, 64  # Increment address for tail because of pad
       vst  v2, a3  # Store extra 512 bits generated by padding
       a0, a0, 1  # Increment N because of extra padding
stripmine:
  vsetvl  t0, a0  # a0 holds veclen N (#512b chunks);t0 ≤ MVL(=4)
vld  v1, a2  # Get first/next 4*512b (256B) of message
vcrypt.hash.SHA256  v3, v1  # compress (4 compressions w/64 rounds each!)
sll  t1, t0, 6  # Multiply count x64 to get byte-delta
add  a1, t1  # Bump address pointers by byte-delta
sub  a0, t0  # subtract number of elements done from N
bnez  a0, stripmine  # Loop if more message to process
vstr  v3, a5  # Save 256b (32B) digest result
```

© 2014 Microsemi Corporation. COMPANY PROPRIETARY
Case Study: WalnutDSA™ Signature Verification

- SecureRF’s Group-Theoretic-Cryptography-based digital signature algorithm
- Allows a signer with a fixed private/public key pair to create a digital signature associated to a given message which can be validated by anyone who knows the public key of the signer
- Main operation performed is “E-Multiplication”, which primarily consists of matrix multiplication in the binary extension field $\text{GF}_{2^5}$
- Multiplicand matrix in E-Multiplication is known as a Colored-Burau (CB) matrix
  - Derived from a braid generator $b_i$ and a T-value (i.e. a given value in the finite field) $t_i$

\[
CB(b_i) = \begin{pmatrix}
1 & \vdots & \vdots & \vdots \\
& t_i & -t_i & 1 \\
& \vdots & \ddots & \vdots \\
& & \ddots & \ddots \\
& & & 1
\end{pmatrix}, \quad CB(b_i^{-1}) = \begin{pmatrix}
1 & \vdots & \vdots & \vdots \\
& \frac{1}{t_i+1} & \frac{1}{t_i+1} & 1 \\
& \vdots & \ddots & \vdots \\
& \ddots & \ddots & \ddots \\
& \vdots & \vdots & \vdots & 1
\end{pmatrix}
\]

- Signature verification metrics presented are based on 128-bit security level (similar to ECC P256 and RSA3072) and are independent of the message hashing operation
Case Study: WalnutDSA Signature Verification

• Colored-Burau’s similarity to the identity matrix allows for optimizations
• Given columns $a$, $b$, $c$ of the multiplier matrix, where $a$, $b$, $c$ correspond to $i-1$, $i$, $i+1$ respectively, standard $n$-by-$n$ matrix multiplication can be substituted by the following three equations:
  • $c_{k_{\text{final}}} = c_{k_{\text{initial}}} + b_{k_{\text{initial}}} \quad (\text{for } k = 1, 2, \ldots, n)$
  • $b_{k_{\text{final}}} = b_{k_{\text{initial}}} \times t_i \quad (\text{for } k = 1, 2, \ldots, n)$
  • $a_{k_{\text{final}}} = a_{k_{\text{initial}}} + b_{k_{\text{final}}} \quad (\text{for } k = 1, 2, \ldots, n)$
• Repeat for all braid generators
  • The product matrix from one iteration becomes the multiplier matrix to the next—this is an iterative operation
• Vector extensions allow for two sources of optimization
  • (a) Parallelize the three equations such that each column element $k$ is operated simultaneously
  • (b) Use vector registers to limit the number of load/store instructions; and reduce instruction bandwidth, in general
• Implementation
  • Load each row of the column-major multiplier matrix into a unique vector register (b)
  • Extract the braid generator and corresponding $T$-value (a)
  • Based upon the braid generator, operate on the three vector registers as defined in the above three equations (a)
  • Store each row vector to the column-major multiplier matrix (b)
• Note that the first and last steps of the implementation are executed once; the remainder are looped for each braid generator
Case Study: WalnutDSA Signature Verification

### Without cryptoextensions

<table>
<thead>
<tr>
<th>SECTION</th>
<th>cycles per iteration</th>
<th>iterations</th>
<th>total cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit</td>
<td>4</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>walnut_emul</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>loop</td>
<td>32</td>
<td>1704</td>
<td>54532</td>
</tr>
<tr>
<td>positive</td>
<td>5</td>
<td>1704</td>
<td>7668</td>
</tr>
<tr>
<td>negative</td>
<td>5</td>
<td>1704</td>
<td>7668</td>
</tr>
<tr>
<td>continue</td>
<td>106</td>
<td>1704</td>
<td>180624</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td></td>
<td></td>
<td><strong>250502</strong></td>
</tr>
</tbody>
</table>

### With cryptoextensions

<table>
<thead>
<tr>
<th>SECTION</th>
<th>cycles per iteration</th>
<th>iterations</th>
<th>total cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit</td>
<td>18</td>
<td>2</td>
<td>36</td>
</tr>
<tr>
<td>walnut_emul</td>
<td>17</td>
<td>2</td>
<td>34</td>
</tr>
<tr>
<td>loop</td>
<td>30</td>
<td>1704</td>
<td>51124</td>
</tr>
<tr>
<td>positive</td>
<td>4</td>
<td>1704</td>
<td>6816</td>
</tr>
<tr>
<td>negative</td>
<td>4</td>
<td>1704</td>
<td>6816</td>
</tr>
<tr>
<td>continue</td>
<td>7</td>
<td>1704</td>
<td>11928</td>
</tr>
<tr>
<td>emultiply</td>
<td>5</td>
<td>1704</td>
<td>8520</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td></td>
<td></td>
<td><strong>85274</strong></td>
</tr>
</tbody>
</table>

- Nearly a 3x speedup with crypto extensions vs. without crypto extensions
- Extra cycles in exit and walnut_emul are due to the loading and storing of the vectors registers, respectively
- With crypto extensions, continue is split into continue and emultiply. The latter is structured like a jump table, allowing for the easy selection and operation of the three vector registers without conditional branching.

2 functions calls due to two distinct braids: encoded message and signature
1704 braid generator estimate (1000 per signature + 704 per encoded message)
Fast Secure Boot – Putting SHA-256 and WalnutDSA together

1 Using letter “Y” for the RISC-V Cryptographic Extensions (not approved yet)