Efficient Side-Channel Protections of ARX Ciphers

Bernhard Jungk\textsuperscript{1} \hspace{1cm} \textbf{Richard Petri}\textsuperscript{2} \hspace{1cm} Marc Stöttinger\textsuperscript{3}

\textsuperscript{1}Fraunhofer Singapore, Singapore, bernhard.jungk@fraunhofer.sg

\textsuperscript{2}Fraunhofer SIT, Germany, richard.petri@sit.fraunhofer.de

\textsuperscript{3}Continental AG, Germany, marc.stoettinger@continental-corporation.com

September 10, 2018
Protecting ARX Ciphers

ARX ciphers (e.g. Threefish, Speck, ChaCha20) rely on modular Addition, Rotation and XOR.
Protecting ARX Ciphers

- ARX ciphers (e.g. Threefish, Speck, ChaCha20) rely on modular Addition, Rotation and XOR
- Easily protected against timing side-channels, but all the harder to protect against power/EM side-channels, see e.g.

![Diagram of ARX cipher operations]
Protecting ARX Ciphers

- ARX ciphers (e.g. Threefish, Speck, ChaCha20) rely on modular **Addition**, **Rotation** and **XOR**

- Easily protected against timing side-channels, but all the harder to protect against power/EM side-channels, see e.g.
  - “Butterfly Attack” against modular addition in Skein
  - “Bricklayer Attack” on ChaCha20
Protecting ARX Ciphers

- ARX ciphers (e.g. Threefish, Speck, ChaCha20) rely on modular Addition, Rotation and XOR
- Easily protected against timing side-channels, but all the harder to protect against power/EM side-channels, see e.g.
  - “Butterfly Attack” against modular addition in Skein
  - “Bricklayer Attack” on ChaCha20
- Early work by Goubin (2001) suggested Boolean and arithmetic masking, with conversion in-between
Protecting ARX Ciphers

- ARX ciphers (e.g. Threefish, Speck, ChaCha20) rely on modular Addition, Rotation and XOR
- Easily protected against timing side-channels, but all the harder to protect against power/EM side-channels, see e.g.
  - “Butterfly Attack” against modular addition in Skein
  - “Bricklayer Attack” on ChaCha20
- Early work by Goubin (2001) suggested Boolean and arithmetic masking, with conversion in-between (Cost: $O(k)$)
Protecting ARX Ciphers

- ARX ciphers (e.g. Threefish, Speck, ChaCha20) rely on modular Addition, Rotation and XOR
- Easily protected against timing side-channels, but all the harder to protect against power/EM side-channels, see e.g.
  - “Butterfly Attack” against modular addition in Skein
  - “Bricklayer Attack” on ChaCha20
- Early work by Goubin (2001) suggested Boolean and arithmetic masking, with conversion in-between (Cost: $O(k)$)
- Simpler: Apply Boolean masking directly to an Addition algorithm in software!
Our contribution

- Threshold Implementations (TI) initially only of interest for hardware implementations until recent developments reduced the number of necessary shares

- Introduce some optimizations for masking additions

- Introduce masked versions of combined SHIFT-AND(-XOR) gates

- Include the "flexible second operand" of ARM platform, performing $z \leftarrow x (y \ll c)$ in one instruction

- Reduce the number of necessary remasking steps, reducing amount of required entropy

- Not in this presentation: We introduce a simpler algorithm for modular subtraction
Our contribution

- Threshold Implementations (TI) initially only of interest for hardware implementations until recent developments reduced the number of necessary shares
- We introduce some optimizations for masking additions
- Introduce masked versions of combined SHIFT-AND(-XOR) gates
- Include the “flexible second operand” of ARM platform, performing \( z \leftarrow x (y \ll c) \) in one instruction
- Reduce the number of necessary remasking steps, reducing amount of required entropy
- Not in this presentation: We introduce a simpler algorithm for modular subtraction
Our contribution

- Threshold Implementations (TI) initially only of interest for hardware implementations until recent developments reduced the number of necessary shares.
- We introduce some optimizations for masking additions.
  - Introduce masked versions of combined SHIFT-AND(-XOR) gates.
  - Include the "flexible second operand" of ARM platform, performing $z \leftarrow x(y \ll c)$ in one instruction.
- Reduce the number of necessary remasking steps, reducing amount of required entropy.
- Not in this presentation: We introduce a simpler algorithm for modular subtraction.
Our contribution

- Threshold Implementations (TI) initially only of interest for hardware implementations until recent developments reduced the number of necessary shares.
- We introduce some optimizations for masking additions:
  - Introduce masked versions of combined SHIFT-AND(-XOR) gates.
  - Include the “flexible second operand” of ARM platform, performing $z \leftarrow x(y \ll c)$ in one instruction.
- Not in this presentation: We introduce a simpler algorithm for modular subtraction.
Our contribution

- Threshold Implementations (TI) initially only of interest for hardware implementations until recent developments reduced the number of necessary shares
- We introduce some optimizations for masking additions
  - Introduce masked versions of combined SHIFT-AND(-XOR) gates
  - Include the “flexible second operand” of ARM platform, performing $z \leftarrow x(y \ll c)$ in one instruction
  - Reduce the number of necessary remasking steps, reducing amount of required entropy
Our contribution

▶ Threshold Implementations (TI) initially only of interest for hardware implementations until recent developments reduced the number of necessary shares
▶ We introduce some optimizations for masking additions
  ▶ Introduce masked versions of combined SHIFT-AND(-XOR) gates
  ▶ Include the “flexible second operand” of ARM platform, performing $z \leftarrow x(y \ll c)$ in one instruction
  ▶ Reduce the number of necessary remasking steps, reducing amount of required entropy
▶ Not in this presentation: We introduce a simpler algorithm for modular subtraction
Kogge-Stone Adder (KSA)

\[
\begin{align*}
\text{Bit 0} & \leftarrow (x[0], y[0]) \\
\text{Iteration 1} & \leftarrow (g[b], p[b]) \\
\text{Iteration 2} & \leftarrow (g[b], g[b]) \\
\text{Iteration 3} & \leftarrow (g[b], p[b]) \\
\end{align*}
\]
Kogge-Stone Adder (KSA)

\[
\begin{align*}
\text{Bit 7: } & (x[7], y[7]) \\
\text{Bit 6: } & (x[6], y[6]) \\
\text{Bit 5: } & (x[5], y[5]) \\
\text{Bit 4: } & (x[4], y[4]) \\
\text{Bit 3: } & (x[3], y[3]) \\
\text{Bit 2: } & (x[2], y[2]) \\
\text{Bit 1: } & (x[1], y[1]) \\
\text{Bit 0: } & (x[0], y[0]) \\
\end{align*}
\]

\[
\begin{align*}
\text{Iteration 1: } & \quad g[b] \leftarrow x[b] \oplus y[b] \\
& \quad p[b] \leftarrow x[b] \land y[b] \\
\text{Iteration 2: } & \quad (g[b], g[b]) \\
& \quad (g[b] \land g[b]) \oplus g[b] \\
& \quad p[b] \leftarrow (p[b] \land g[b]) \\
\text{Iteration 3: } & \quad \quad \quad \quad (g[b], p[b])
\end{align*}
\]

Combined SHIFT-AND(-XOR) gates
TI AND(-XOR) Gate with 2 shares

\[(z_0 \oplus z_1) \leftarrow (x_0 \oplus x_1) \land (y_0 \oplus y_1)\]

\[s_0 \leftarrow x_0 \land y_0, \quad s_1 \leftarrow x_0 \land y_1\]
\[s_2 \leftarrow x_1 \land y_0, \quad s_3 \leftarrow x_1 \land y_1\]
\[z_0 \leftarrow s_0 \oplus s_2, \quad z_1 \leftarrow s_1 \oplus s_3\]

- Direct approach to constructing an AND gate with four output shares, which are registered and recombined.
TI AND(-XOR) Gate with 2 shares

\[(z_0 \oplus z_1) \leftarrow (x_0 \oplus x_1) \land (y_0 \oplus y_1)\]

\[s_0 \leftarrow x_0 \land y_0, \quad s_1 \leftarrow x_0 \land y_1\]
\[s_2 \leftarrow x_1 \land y_0, \quad s_3 \leftarrow x_1 \land y_1\]
\[t_0 \leftarrow s_0 \oplus m, \quad t_1 \leftarrow s_1 \oplus m\]
\[z_0 \leftarrow t_0 \oplus s_2, \quad z_1 \leftarrow t_1 \oplus s_3\]

- Direct approach to constructing an AND gate with four output shares, which are registered and recombined
- Output is not uniform, requiring remasking with a guard share \(m\)
TI AND(-XOR) Gate with 2 shares

\[(z_0 \oplus z_1) \leftarrow (x_0 \oplus x_1) \land (y_0 \oplus y_1)\]

\[m \leftarrow (x_0 \gg 1) \oplus (u \ll k - 1)\]

\[s_0 \leftarrow x_0 \land y_0, \quad s_1 \leftarrow x_0 \land y_1\]

\[s_2 \leftarrow x_1 \land y_0, \quad s_3 \leftarrow x_1 \land y_1\]

\[t_0 \leftarrow s_0 \oplus m, \quad t_1 \leftarrow s_1 \oplus m\]

\[z_0 \leftarrow t_0 \oplus s_2, \quad z_1 \leftarrow t_1 \oplus s_3\]

- Direct approach to constructing an AND gate with four output shares, which are registered and recombined
- Output is not uniform, requiring remasking with a guard share \(m\)
- Typical software implementation processes \(k\)-shares in parallel \(\rightarrow\) use one uniform input shares as guard share (just need one fresh bit)
**TI AND(-XOR) Gate with 2 shares**

\[(z_0 \oplus z_1) \leftarrow (x_0 \oplus x_1) \land (y_0 \oplus y_1) \oplus (u_0 \oplus u_1)\]

\[
\begin{align*}
s_0 & \leftarrow x_0 \land y_0, \\
s_1 & \leftarrow x_0 \land y_1 \\
s_2 & \leftarrow x_1 \land y_0, \\
s_3 & \leftarrow x_1 \land y_1 \\
t_0 & \leftarrow s_0 \oplus u_0, \\
t_1 & \leftarrow s_1 \oplus u_1 \\
z_0 & \leftarrow t_0 \oplus s_2, \\
z_1 & \leftarrow t_1 \oplus s_3
\end{align*}
\]

- Direct approach to constructing an AND gate with four output shares, which are registered and recombeded.
- Output is not uniform, requiring remasking with a guard share \(m\).
- Typical software implementation processes \(k\)-shares in parallel \(\rightarrow\) use one uniform input shares as guard share (just need one fresh bit).
- In the case of \(z \leftarrow (x \land y) \oplus u\) no guard share is required.
Combined SHIFT-AND(-XOR) gate

\[ m \leftarrow (x_0 \gg 1) \oplus (u \ll k - 1) \]
\[ s_0 \leftarrow x_0 \land (x_0 \ll i), \quad s_1 \leftarrow x_0 \land (x_1 \ll i) \]
\[ s_2 \leftarrow x_1 \land (x_0 \ll i), \quad s_3 \leftarrow x_1 \land (x_1 \ll i) \]
\[ t_0 \leftarrow s_0 \oplus m, \quad t_1 \leftarrow s_1 \oplus m \]
\[ z_0 \leftarrow t_0 \oplus s_2, \quad z_1 \leftarrow t_1 \oplus s_3 \]

The KSA heavily uses a combined SHIFT-AND (and SHIFT-AND-XOR) operation which lends itself well to the ARM “flexible second operand”
Combined SHIFT-AND(-XOR) gate

\[ s_0 \leftarrow x_0 \land (y_0 \ll i), \]
\[ s_2 \leftarrow x_1 \land (y_0 \ll i), \]
\[ t_0 \leftarrow s_0 \oplus y_0, \]
\[ z_0 \leftarrow t_0 \oplus s_2, \]
\[ s_1 \leftarrow x_0 \land (y_1 \ll i) \]
\[ s_3 \leftarrow x_1 \land (y_1 \ll i) \]
\[ t_1 \leftarrow s_1 \oplus y_1 \]
\[ z_1 \leftarrow t_1 \oplus s_3 \]

- The KSA heavily uses a combined SHIFT-AND (and SHIFT-AND-XOR) operation which lends itself well to the ARM “flexible second operand”
- Again, in the case of \( z \leftarrow (x \land (y << i)) \oplus y \) no guard share is required
Protected KSA

Require: $x, y \in \mathbb{Z}_{2^k}, k > 0$

Ensure: $z = (x + y) \mod 2^k$

1: $n \leftarrow \max(\lceil \log_2(k - 1) \rceil, 1)$
2: $g \leftarrow x \land y$
3: $p \leftarrow x \oplus y$

4: for $i = 1$ to $n - 1$ do

5: $g \leftarrow (p \land (g \ll 2^{i-1})) \oplus g$
6: $p \leftarrow (p \land (p \ll 2^{i-1}))$

7: end for
8: $g \leftarrow (p \land (g \ll 2^{n-1})) \oplus g$
9: $z \leftarrow x \oplus y \oplus 2g$
10: return $z$
Protected KSA

Require: \(x_0, x_1, y_0, y_1 \in \mathbb{Z}_{2^k}, k > 0, u \in \{0, 1\}\), with \(x = x_0 \oplus x_1\) and \(y = y_0 \oplus y_1\)

Ensure: \(z = (x + y) \mod 2^k\), with \(z = z_0 \oplus z_1\)

1: \(n \leftarrow \max(\lceil \log_2(k - 1) \rceil, 1)\) # Shared AND
2: \((g_0, g_1) \leftarrow \text{SecAnd}(x_0, x_1, y_0, y_1, u)\) # Shared AND
3: \((p_0, p_1) \leftarrow \text{SecXor}(x_0, x_1, y_0, y_1)\) # Shared XOR
4: \(u \leftarrow x_0 \mod 2\) # Update guard share
5: for \(i = 1\) to \(n - 1\) do
6: \(v \leftarrow p_0 \mod 2\) # Save next guard share
7: \((g_0, g_1) \leftarrow \text{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{i-1})\) # Shared AND-SHIFT-XOR
8: \((p_0, p_1) \leftarrow \text{SecAndShift}(p_0, p_1, u, 2^{i-1})\) # Shared AND-SHIFT
9: \(u \leftarrow v\) # Update guard share
10: end for
11: \((g_0, g_1) \leftarrow \text{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{n-1})\) # Shared AND-SHIFT-XOR
12: \((z_0, z_1) \leftarrow (x_0 \oplus y_0 \oplus 2g_0, x_1 \oplus y_1 \oplus 2g_1)\) # Compute final output
13: return \((z_0, z_1, u)\)
Protected KSA

Require: \(x_0, x_1, y_0, y_1 \in \mathbb{Z}_{2^k}, \ k > 0, \ u \in \{0, 1\}\), with \(x = x_0 \oplus x_1\) and \(y = y_0 \oplus y_1\)

Ensure: \(z = (x + y) \mod 2^k\), with \(z = z_0 \oplus z_1\)

1: \(n \leftarrow \max\left(\lceil \log_2(k - 1) \rceil, 1\right)\)
2: \((g_0, g_1) \leftarrow \text{SecAnd}(x_0, x_1, y_0, y_1, u)\) \# Shared AND
3: \((p_0, p_1) \leftarrow \text{SecXor}(x_0, x_1, y_0, y_1)\) \# Shared XOR
4: \(u \leftarrow x_0 \mod 2\) \# Update guard share
5: for \(i = 1\) to \(n - 1\) do
6: \(v \leftarrow p_0 \mod 2\) \# Save next guard share
7: \((g_0, g_1) \leftarrow \text{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{i-1})\) \# Shared AND-SHIFT-XOR
8: \((p_0, p_1) \leftarrow \text{SecAndShift}(p_0, p_1, u, 2^{i-1})\) \# Shared AND-SHIFT
9: \(u \leftarrow v\) \# Update guard share
10: end for
11: \((g_0, g_1) \leftarrow \text{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{n-1})\) \# Shared AND-SHIFT-XOR
12: \((z_0, z_1) \leftarrow (x_0 \oplus y_0 \oplus 2g_0, x_1 \oplus y_1 \oplus 2g_1)\) \# Compute final output
13: return \((z_0, z_1, u)\)
Protected KSA

- Bit 0: \((x[0], y[0])\)
- Bit 1: \((x[1], y[1])\)
- Bit 2: \((x[2], y[2])\)
- Bit 3: \((x[3], y[3])\)
- Bit 4: \((x[4], y[4])\)
- Bit 5: \((x[5], y[5])\)
- Bit 6: \((x[6], y[6])\)
- Bit 7: \((x[7], y[7])\)

Input

Iteration 1

LSB can be used as guard share for next iteration

Iteration 3

Output
Protected KSA

Require: $x_0, x_1, y_0, y_1 \in \mathbb{Z}_{2^k}$, $k > 0$, $u \in \{0, 1\}$, with $x = x_0 \oplus x_1$ and $y = y_0 \oplus y_1$
Ensure: $z = (x + y) \mod 2^k$, with $z = z_0 \oplus z_1$

1: $n \leftarrow \max([\log_2(k - 1)], 1)$
2: $(g_0, g_1) \leftarrow \text{SecAnd}(x_0, x_1, y_0, y_1, u)$  \hspace{1cm} # Shared AND
3: $(p_0, p_1) \leftarrow \text{SecXor}(x_0, x_1, y_0, y_1)$  \hspace{1cm} # Shared XOR
4: $u \leftarrow x_0 \mod 2$  \hspace{1cm} # Update guard share
5: for $i = 1$ to $n - 1$ do
6:  \hspace{1cm} $v \leftarrow p_0 \mod 2$  \hspace{1cm} # Save next guard share
7:  \hspace{1cm} $(g_0, g_1) \leftarrow \text{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{i-1})$  \hspace{1cm} # Shared AND-SHIFT-XOR
8:  \hspace{1cm} $(p_0, p_1) \leftarrow \text{SecAndShift}(p_0, p_1, u, 2^{i-1})$  \hspace{1cm} # Shared AND-SHIFT
9:  \hspace{1cm} $u \leftarrow v$  \hspace{1cm} # Update guard share
10: end for
11: $(g_0, g_1) \leftarrow \text{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{n-1})$  \hspace{1cm} # Shared AND-SHIFT-XOR
12: $(z_0, z_1) \leftarrow (x_0 \oplus y_0 \oplus 2g_0, x_1 \oplus y_1 \oplus 2g_1)$  \hspace{1cm} # Compute final output
13: return $(z_0, z_1, u)$
Further optimization

\[
\begin{align*}
  s_0 & \leftarrow x_0 \land y_0, \\
  s_2 & \leftarrow x_1 \land y_0, \\
  z_0 & \leftarrow s_0 \oplus s_1, \\
  s_1 & \leftarrow x_0 \lor \neg y_1, \\
  s_3 & \leftarrow x_1 \lor \neg y_1, \\
  z_1 & \leftarrow s_2 \oplus s_3
\end{align*}
\]

Biryukov et al. (2017) introduced a further optimized secure AND gate (SecAndOpt/SecAndShiftOpt) which can be combined with our approach.
## Comparision of masked operations

<table>
<thead>
<tr>
<th></th>
<th>SecXor</th>
<th>SecShift</th>
<th>SecAnd</th>
<th>SecAndShift / -Opt</th>
<th>SecAndShiftXor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Generic [Coron et al.]</strong></td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>8 + 2</td>
<td>8 + 4 + 2</td>
</tr>
<tr>
<td><strong>ARM [Coron et al.]</strong></td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>8 + 2</td>
<td>8 + 4 + 2</td>
</tr>
<tr>
<td><strong>Generic [Biryukov et al.]</strong></td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>7 + 2</td>
<td>7 + 2 + 2</td>
</tr>
<tr>
<td><strong>ARM [Biryukov et al.]</strong></td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6 + 2</td>
<td>6 + 2 + 2</td>
</tr>
<tr>
<td><strong>Generic [new]</strong></td>
<td>2</td>
<td>-</td>
<td>8</td>
<td>10 / 9</td>
<td>10</td>
</tr>
<tr>
<td><strong>ARM [new]</strong></td>
<td>2</td>
<td>-</td>
<td>8</td>
<td>8 / 6</td>
<td>8</td>
</tr>
</tbody>
</table>

- Combined AND-SHIFT operations save most of the instructions
## Comparision of masked operations

<table>
<thead>
<tr>
<th></th>
<th>SecXor</th>
<th>SecShift</th>
<th>SecAnd</th>
<th>SecAndShift / -Opt</th>
<th>SecAndShiftXor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Generic [Coron et al.]</strong></td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>8 + 2</td>
<td>8 + 4 + 2</td>
</tr>
<tr>
<td>ARM [Coron et al.]</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>8 + 2</td>
<td>8 + 4 + 2</td>
</tr>
<tr>
<td><strong>Generic [Biryukov et al.]</strong></td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>7 + 2</td>
<td>7 + 2 + 2</td>
</tr>
<tr>
<td>ARM [Biryukov et al.]</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6 + 2</td>
<td>6 + 2 + 2</td>
</tr>
<tr>
<td><strong>Generic [new]</strong></td>
<td>2</td>
<td>-</td>
<td>8</td>
<td>10 / 9</td>
<td>10</td>
</tr>
<tr>
<td>ARM [new]</td>
<td>2</td>
<td>-</td>
<td>8</td>
<td>8 / 6</td>
<td>8</td>
</tr>
</tbody>
</table>

- Combined AND-SHIFT operations save most of the instructions
- Especially when combined with optimizations proposed by Biryukov et al.
## Comparision of masked operations

<table>
<thead>
<tr>
<th></th>
<th>SecXor</th>
<th>SecShift</th>
<th>SecAnd</th>
<th>SecAndShift / -Opt</th>
<th>SecAndShiftXor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic [Coron et al.]</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>8 + 2</td>
<td>8 + 4 + 2</td>
</tr>
<tr>
<td>ARM [Coron et al.]</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>8 + 2</td>
<td>8 + 4 + 2</td>
</tr>
<tr>
<td>Generic [Biryukov et al.]</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>7 + 2</td>
<td>7 + 2 + 2</td>
</tr>
<tr>
<td>ARM [Biryukov et al.]</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6 + 2</td>
<td>6 + 2 + 2</td>
</tr>
<tr>
<td>Generic [new]</td>
<td>2</td>
<td>-</td>
<td>8</td>
<td>10 / 9</td>
<td>10</td>
</tr>
<tr>
<td>ARM [new]</td>
<td>2</td>
<td>-</td>
<td>8</td>
<td>8 / 6</td>
<td>8</td>
</tr>
</tbody>
</table>

- Combined AND-SHIFT operations save most of the instructions
- Especially when combined with optimizations proposed by Biryukov et al.
- Generation of refresh mask takes only 3 instructions
Comparison of masked 32-bit modular addition

- ARM implementation improved by 31% when combined with approach by Biryukov et al.

### Instructions

<table>
<thead>
<tr>
<th></th>
<th>Coron et al.</th>
<th>Biryukov et al.</th>
<th>This work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>144</td>
<td>164</td>
<td>156</td>
</tr>
<tr>
<td>Add (ARM)</td>
<td>106</td>
<td>78</td>
<td>83</td>
</tr>
<tr>
<td>Sub</td>
<td>114</td>
<td>112</td>
<td>116</td>
</tr>
<tr>
<td>Sub (ARM)</td>
<td>106</td>
<td>114</td>
<td>116</td>
</tr>
</tbody>
</table>

- Significantly improved subtraction instruction counts
- Needs one random bit, outputs one random bit
<table>
<thead>
<tr>
<th></th>
<th>Coron et al.</th>
<th>Biryukov et al.</th>
<th>This work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>144</td>
<td>116</td>
<td>106</td>
</tr>
<tr>
<td>Add (ARM)</td>
<td>144</td>
<td>114</td>
<td>78</td>
</tr>
<tr>
<td>Sub</td>
<td>164</td>
<td>112</td>
<td>83</td>
</tr>
<tr>
<td>Sub (ARM)</td>
<td>156</td>
<td></td>
<td>83</td>
</tr>
</tbody>
</table>

ARM implementation improved by 31% when combined with approach by Biryukov et al.

Significantly improved subtraction instruction counts
Comparision of masked 32-bit modular addition

<table>
<thead>
<tr>
<th></th>
<th>Coron et al.</th>
<th>Biryukov et al.</th>
<th>This work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>144</td>
<td>144</td>
<td>116</td>
</tr>
<tr>
<td>Add (ARM)</td>
<td>144</td>
<td>114</td>
<td>78</td>
</tr>
<tr>
<td>Sub</td>
<td>164</td>
<td>112</td>
<td>156</td>
</tr>
<tr>
<td>Sub (ARM)</td>
<td>164</td>
<td>112</td>
<td>83</td>
</tr>
</tbody>
</table>

- ARM implementation improved by 31% when combined with approach by Biryukov et al.
- Significantly improved subtraction instruction counts
- Needs one random bit, outputs one random bit
Application to ChaCha20 cipher

- We implemented an unprotected reference and two protected variants

<table>
<thead>
<tr>
<th></th>
<th>Reference</th>
<th>Previous Results</th>
<th>This Work</th>
</tr>
</thead>
<tbody>
<tr>
<td># cycles</td>
<td>1,726</td>
<td>121,618</td>
<td>72,721</td>
</tr>
<tr>
<td></td>
<td></td>
<td>93,993</td>
<td>60,623</td>
</tr>
</tbody>
</table>

- Masked [Adomnicai et al.]
- Masked Opt. [Adomnicai et al.]

- TI 2-share
- TI 2-share Opt.
Application to ChaCha20 cipher

- We implemented an unprotected reference and two protected variants
- Masked addition is the driving factor

<table>
<thead>
<tr>
<th></th>
<th># cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>1,726</td>
</tr>
<tr>
<td>Previous Results</td>
<td>121,618</td>
</tr>
<tr>
<td>This Work</td>
<td>72,721</td>
</tr>
</tbody>
</table>

Reference

- Previous Results
- This Work

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># cycles</td>
<td>93,993</td>
<td>60,623</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>TI 2-share</th>
<th>TI 2-share Opt.</th>
</tr>
</thead>
<tbody>
<tr>
<td># cycles</td>
<td>72,721</td>
<td>60,623</td>
</tr>
</tbody>
</table>
We implemented an unprotected reference and two protected variants. Masked addition is the driving factor. Note: cycle-counts not entirely comparable due to possible differences in memory architecture.

<table>
<thead>
<tr>
<th></th>
<th>Reference</th>
<th>Previous Results</th>
<th>This Work</th>
</tr>
</thead>
<tbody>
<tr>
<td># cycles</td>
<td>1,726</td>
<td>121,618</td>
<td>93,993</td>
</tr>
<tr>
<td></td>
<td></td>
<td>72,721</td>
<td>60,623</td>
</tr>
</tbody>
</table>

- Masked [Adomnicai et al.]
- Masked Opt. [Adomnicai et al.]
- TI 2-share
- TI 2-share Opt.
Simulation

- ChaCha implementation was simulated with Micro-Architectural Power Simulator (MAPS)\(^1\)
- Simulator was extended by 11 instructions
- Hamming distance is sampled for each register assignment
- \(t\)-Test with a fixed vs. random setup and \(10^5\) noise free traces
- Noise amplification methods like shuffling should still be used

---

\(^1\)https://github.com/cryptolu/maps
Thank you for listening
In the case of ChaCha, shuffling can be used to amplify the noise.

ChaCha State consists of 4 columns which are processed independently (within a round).

Instead of processing columns sequentially, one can jump between columns.

\[
\frac{(4\cdot12)!}{(12!)^4} \approx 2^{88} \text{ Permutations}
\]

Noise can be further amplified by splitting the masked addition into several operations.
In the case of ChaCha, shuffling can be used to amplify the noise.

ChaCha State consists of 4 columns which are processed independently (within a round).

Instead of processing columns sequentially, one can jump between columns.

\[
\frac{(4\cdot12)!}{(12!)^4} \approx 2^{88} \text{ Permutations}
\]

Noise can be further amplified by splitting the masked addition into several operations.