# Efficient Side-Channel Protections of ARX Ciphers

Bernhard Jungk[1] **Richard Petri**[2] Marc Stöttinger[3]

[1]Fraunhofer Singapore, Singapore, bernhard.jungk@fraunhofer.sg

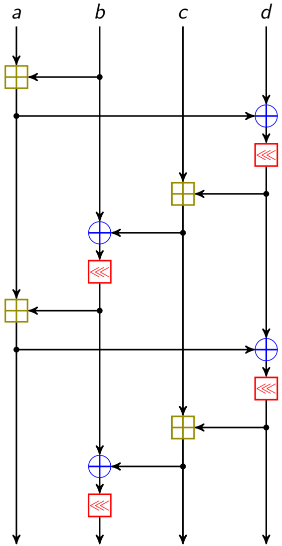
[2]Fraunhofer SIT, Germany, richard.petri@sit.fraunhofer.de

[3]Continental AG, Germany, marc.stoettinger@contiental-corporation.com

September 10, 2018

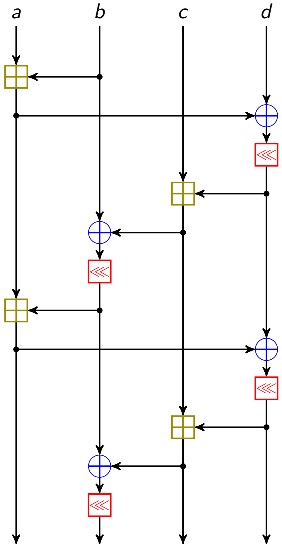
# Protecting ARX Ciphers

- ARX ciphers (e.g. Threefish, Speck, ChaCha20) rely on modular **A**ddition, **R**otation and **X**OR



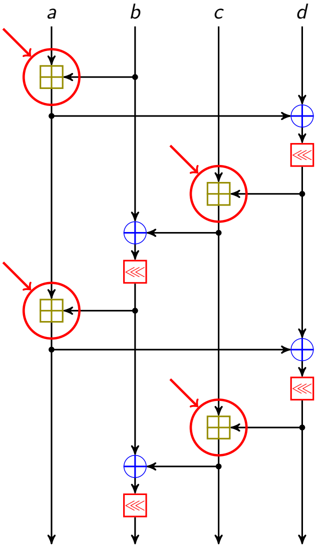
# Protecting ARX Ciphers

- ARX ciphers (e.g. Threefish, Speck, ChaCha20) rely on modular **A**ddition, **R**otation and **X**OR
- Easily protected against timing side-channels, but all the harder to protect against power/EM side-channels, see e.g.



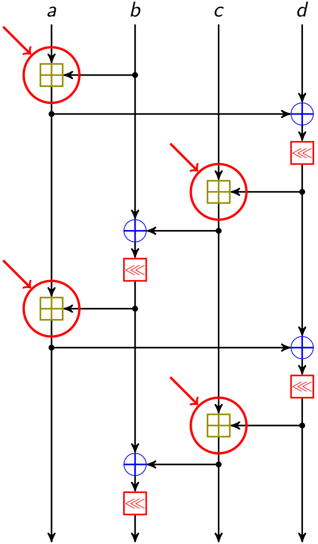
# Protecting ARX Ciphers

- ARX ciphers (e.g. Threefish, Speck, ChaCha20) rely on modular **A**ddition, **R**otation and **X**OR
- 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

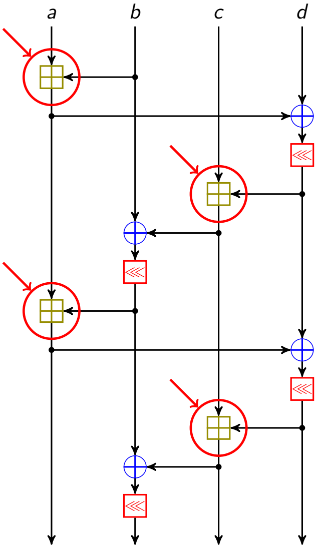


# Protecting ARX Ciphers

- ARX ciphers (e.g. Threefish, Speck, ChaCha20) rely on modular **A**ddition, **R**otation and **X**OR
- 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



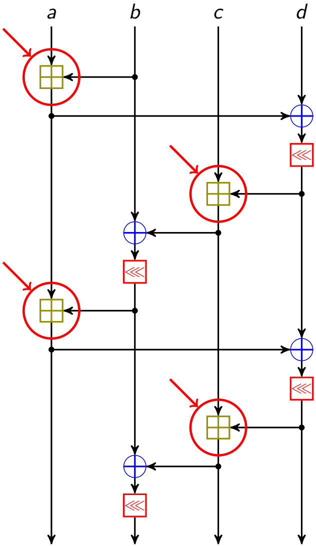
# Protecting ARX Ciphers



- ARX ciphers (e.g. Threefish, Speck, ChaCha20) rely on modular **A**ddition, **R**otation and **X**OR
- 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: $\mathcal{O}(k)$)

# Protecting ARX Ciphers

- ARX ciphers (e.g. Threefish, Speck, ChaCha20) rely on modular **A**ddition, **R**otation and **X**OR
- 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: $\mathcal{O}(k)$)
- 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

# 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

# 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

# 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

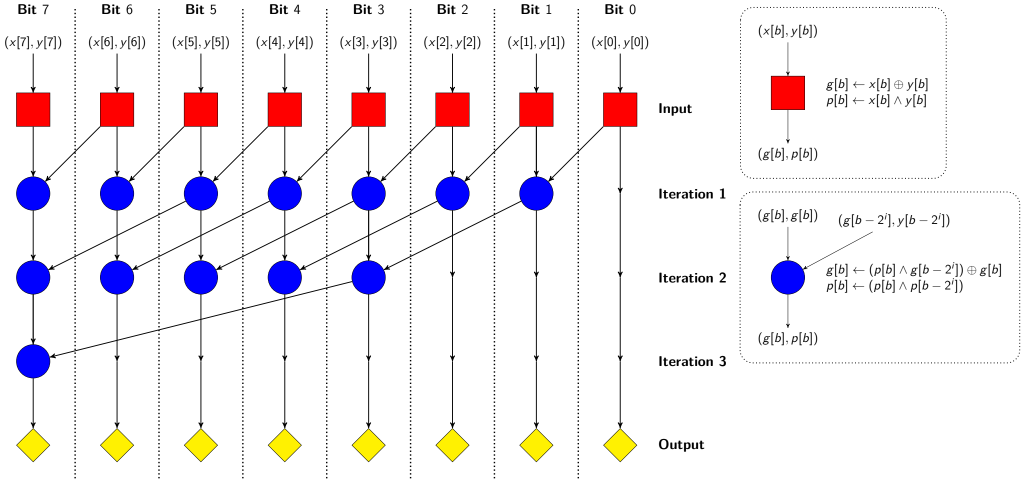
# Our contribution

- 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)



$(x[b], y[b])$

$g[b] \leftarrow x[b] \oplus y[b]$
$p[b] \leftarrow x[b] \wedge y[b]$

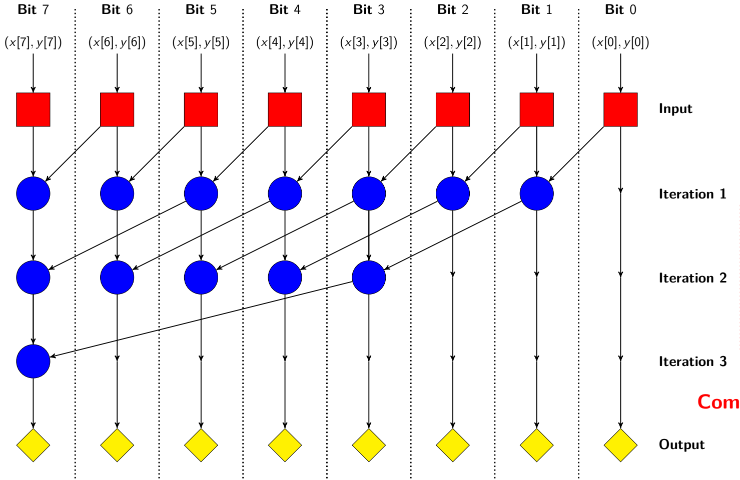
$(g[b], p[b])$

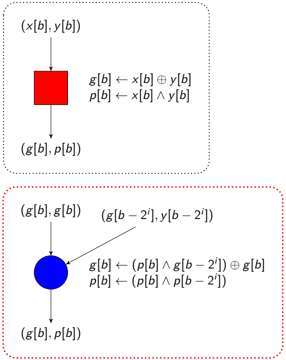
$(g[b], g[b])$ $(g[b - 2^i], y[b - 2^i])$

$g[b] \leftarrow (p[b] \wedge g[b - 2^i]) \oplus g[b]$
$p[b] \leftarrow (p[b] \wedge p[b - 2^i])$

$(g[b], p[b])$

# Kogge-Stone Adder (KSA)





**Combined SHIFT-AND(-XOR) gates**

# TI AND(-XOR) Gate with 2 shares

$$(z_0 \oplus z_1) \leftarrow (x_0 \oplus x_1) \wedge (y_0 \oplus y_1)$$

$$\begin{aligned} s_0 &\leftarrow x_0 \wedge y_0, & s_1 &\leftarrow x_0 \wedge y_1 \\ s_2 &\leftarrow x_1 \wedge y_0, & s_3 &\leftarrow x_1 \wedge y_1 \\ z_0 &\leftarrow s_0 \oplus s_2, & z_1 &\leftarrow s_1 \oplus s_3 \end{aligned}$$

- Direct approach to constructing an AND gate with four output shares, which are registered and recombined

# TI AND(-XOR) Gate with 2 shares

$$(z_0 \oplus z_1) \leftarrow (x_0 \oplus x_1) \wedge (y_0 \oplus y_1)$$

$$\begin{aligned}
s_0 &\leftarrow x_0 \wedge y_0, & s_1 &\leftarrow x_0 \wedge y_1 \\
s_2 &\leftarrow x_1 \wedge y_0, & s_3 &\leftarrow x_1 \wedge y_1 \\
t_0 &\leftarrow s_0 \oplus m, & t_1 &\leftarrow s_1 \oplus m \\
z_0 &\leftarrow t_0 \oplus s_2, & z_1 &\leftarrow t_1 \oplus s_3
\end{aligned}$$

- 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

$$(z_0 \oplus z_1) \leftarrow (x_0 \oplus x_1) \wedge (y_0 \oplus y_1)$$

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

$$\begin{aligned} s_0 &\leftarrow x_0 \wedge y_0, & s_1 &\leftarrow x_0 \wedge y_1 \\ s_2 &\leftarrow x_1 \wedge y_0, & s_3 &\leftarrow x_1 \wedge y_1 \\ t_0 &\leftarrow s_0 \oplus m, & t_1 &\leftarrow s_1 \oplus m \\ z_0 &\leftarrow t_0 \oplus s_2, & z_1 &\leftarrow t_1 \oplus s_3 \end{aligned}$$

- 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 → use one uniform input shares as guard share (just need one fresh bit)

# TI AND(-XOR) Gate with 2 shares

$$(z_0 \oplus z_1) \leftarrow (x_0 \oplus x_1) \wedge (y_0 \oplus y_1) \oplus (u_0 \oplus u_1)$$

$$\begin{aligned} s_0 &\leftarrow x_0 \wedge y_0, & s_1 &\leftarrow x_0 \wedge y_1 \\ s_2 &\leftarrow x_1 \wedge y_0, & s_3 &\leftarrow x_1 \wedge y_1 \\ t_0 &\leftarrow s_0 \oplus u_0, & t_1 &\leftarrow s_1 \oplus u_1 \\ z_0 &\leftarrow t_0 \oplus s_2, & z_1 &\leftarrow t_1 \oplus s_3 \end{aligned}$$

- 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 → use one uniform input shares as guard share (just need one fresh bit)
- In the case of $z \leftarrow (x \wedge y) \oplus u$ no guard share is required

# Combined SHIFT-AND(-XOR) gate

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

$$\begin{aligned} s_0 &\leftarrow x_0 \wedge (x_0 \ll i), & s_1 &\leftarrow x_0 \wedge (x_1 \ll i) \\ s_2 &\leftarrow x_1 \wedge (x_0 \ll i), & s_3 &\leftarrow x_1 \wedge (x_1 \ll i) \\ t_0 &\leftarrow s_0 \oplus m, & t_1 &\leftarrow s_1 \oplus m \\ z_0 &\leftarrow t_0 \oplus s_2, & z_1 &\leftarrow t_1 \oplus s_3 \end{aligned}$$

- 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

$$\begin{aligned} s_0 &\leftarrow x_0 \wedge (y_0 \ll i), \\ s_2 &\leftarrow x_1 \wedge (y_0 \ll i), \\ t_0 &\leftarrow s_0 \oplus \color{red}{y_0}, \\ z_0 &\leftarrow t_0 \oplus s_2, \end{aligned} \qquad \begin{aligned} s_1 &\leftarrow x_0 \wedge (y_1 \ll i) \\ s_3 &\leftarrow x_1 \wedge (y_1 \ll i) \\ t_1 &\leftarrow s_1 \oplus \color{red}{y_1} \\ z_1 &\leftarrow t_1 \oplus s_3 \end{aligned}$$

- 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 $z \leftarrow (x \wedge (y << i)) \oplus y$ no guard share is required

# Protected KSA

```
Require: $x, y \in \mathbb{Z}_{2^k},\ k > 0$
Ensure: $z = (x + y) \bmod 2^k$
$n \leftarrow \max(\lceil \log_2(k - 1) \rceil, 1)$
$g \leftarrow x \wedge y$
$p \leftarrow x \oplus y$

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

    $g \leftarrow (p \wedge (g \ll 2^{i-1})) \oplus g$
    $p \leftarrow (p \wedge (p \ll 2^{i-1}))$

end for
$g \leftarrow (p \wedge (g \ll 2^{n-1})) \oplus g$
$z \leftarrow x \oplus y \oplus 2g$
return $z$
```

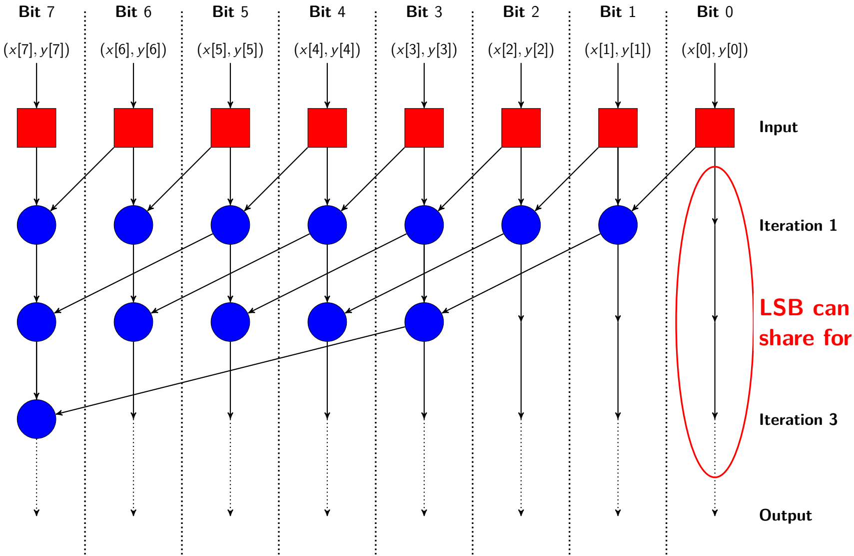
# Protected KSA

```
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: $z = (x + y) \bmod 2^k$, with $z = z_0 \oplus z_1$
$n \leftarrow \max(\lceil \log_2(k - 1) \rceil, 1)$
$(g_0, g_1) \leftarrow \texttt{SecAnd}(x_0, x_1, y_0, y_1, u)$                # Shared AND
$(p_0, p_1) \leftarrow \texttt{SecXor}(x_0, x_1, y_0, y_1)$                   # Shared XOR
$u \leftarrow x_0 \bmod 2$                                                    # Update guard share
for $i = 1$ to $n - 1$ do
    $v \leftarrow p_0 \bmod 2$                                                # Save next guard share
    $(g_0, g_1) \leftarrow \texttt{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{i-1})$   # Shared AND-SHIFT-XOR
    $(p_0, p_1) \leftarrow \texttt{SecAndShift}(p_0, p_1, u, 2^{i-1})$        # Shared AND-SHIFT
    $u \leftarrow v$                                                          # Update guard share
end for
$(g_0, g_1) \leftarrow \texttt{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{n-1})$       # Shared AND-SHIFT-XOR
$(z_0, z_1) \leftarrow (x_0 \oplus y_0 \oplus 2g_0, x_1 \oplus y_1 \oplus 2g_1)$   # Compute final output
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: $z = (x + y) \bmod 2^k$, with $z = z_0 \oplus z_1$
$n \leftarrow \max(\lceil \log_2(k-1) \rceil, 1)$
$(g_0, g_1) \leftarrow \texttt{SecAnd}(x_0, x_1, y_0, y_1, u)$                # Shared AND
$(p_0, p_1) \leftarrow \texttt{SecXor}(x_0, x_1, y_0, y_1)$                   # Shared XOR
$u \leftarrow x_0 \bmod 2$                                                    # Update guard share
for $i = 1$ to $n - 1$ do
    $v \leftarrow p_0 \bmod 2$                                                # Save next guard share
    $(g_0, g_1) \leftarrow \texttt{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{i-1})$   # Shared AND-SHIFT-XOR
    $(p_0, p_1) \leftarrow \texttt{SecAndShift}(p_0, p_1, u, 2^{i-1})$        # Shared AND-SHIFT
    $u \leftarrow v$                                                          # Update guard share
end for
$(g_0, g_1) \leftarrow \texttt{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{n-1})$       # Shared AND-SHIFT-XOR
$(z_0, z_1) \leftarrow (x_0 \oplus y_0 \oplus 2g_0, x_1 \oplus y_1 \oplus 2g_1)$   # Compute final output
return $(z_0, z_1, u)$
```

# Protected KSA



**LSB can be used as guard share for next iteration**

# 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: $z = (x + y) \bmod 2^k$, with $z = z_0 \oplus z_1$
$n \leftarrow \max(\lceil \log_2(k-1) \rceil, 1)$
$(g_0, g_1) \leftarrow \texttt{SecAnd}(x_0, x_1, y_0, y_1, u)$                # Shared AND
$(p_0, p_1) \leftarrow \texttt{SecXor}(x_0, x_1, y_0, y_1)$                   # Shared XOR
$u \leftarrow x_0 \bmod 2$                                                     # Update guard share
for $i = 1$ to $n - 1$ do
    $v \leftarrow p_0 \bmod 2$                                                 # Save next guard share
    $(g_0, g_1) \leftarrow \texttt{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{i-1})$   # Shared AND-SHIFT-XOR
    $(p_0, p_1) \leftarrow \texttt{SecAndShift}(p_0, p_1, u, 2^{i-1})$             # Shared AND-SHIFT
    $u \leftarrow v$                                                           # Update guard share
end for
$(g_0, g_1) \leftarrow \texttt{SecAndShiftXor}(p_0, p_1, g_0, g_1, 2^{n-1})$       # Shared AND-SHIFT-XOR
$(z_0, z_1) \leftarrow (x_0 \oplus y_0 \oplus 2g_0, x_1 \oplus y_1 \oplus 2g_1)$  # Compute final output
return $(z_0, z_1, u)$
```

# Further optimization

$$\begin{aligned} s_0 &\leftarrow x_0 \wedge y_0, & s_1 &\leftarrow x_0 \vee \neg y_1 \\ s_2 &\leftarrow x_1 \wedge y_0, & s_3 &\leftarrow x_1 \vee \neg y_1 \\ z_0 &\leftarrow s_0 \oplus s_1, & z_1 &\leftarrow s_2 \oplus s_3 \end{aligned}$$

- Biryukov et al. (2017) introduced a further optimized secure AND gate (`SecAndOpt`/`SecAndShiftOpt`) which can be combined with our approach

# Comparision of masked operations

| | `SecXor` | `SecShift` | `SecAnd` | `SecAndShift` / `-Opt` | `SecAndShiftXor` |
|---|---|---|---|---|---|
| Generic [Coron et al.] | 2 | 4 | 8 | $8 + 2$ | $8 + 4 + 2$ |
| ARM [Coron et al.] | 2 | 4 | 8 | $8 + 2$ | $8 + 4 + 2$ |
| Generic [Biryukov et al.] | 2 | 2 | 7 | $7 + 2$ | $7 + 2 + 2$ |
| ARM [Biryukov et al.] | 2 | 2 | 6 | $6 + 2$ | $6 + 2 + 2$ |
| Generic [new] | 2 | - | 8 | 10 / 9 | 10 |
| ARM [new] | 2 | - | 8 | 8 / 6 | 8 |

- Combined AND-SHIFT operations save most of the instructions

# Comparision of masked operations

| | `SecXor` | `SecShift` | `SecAnd` | `SecAndShift` / `-Opt` | `SecAndShiftXor` |
|---|---|---|---|---|---|
| Generic [Coron et al.] | 2 | 4 | 8 | $8+2$ | $8+4+2$ |
| ARM [Coron et al.] | 2 | 4 | 8 | $8+2$ | $8+4+2$ |
| Generic [Biryukov et al.] | 2 | 2 | 7 | $7+2$ | $7+2+2$ |
| ARM [Biryukov et al.] | 2 | 2 | 6 | $6+2$ | $6+2+2$ |
| Generic [new] | 2 | - | 8 | 10 / 9 | 10 |
| ARM [new] | 2 | - | 8 | 8 / 6 | 8 |

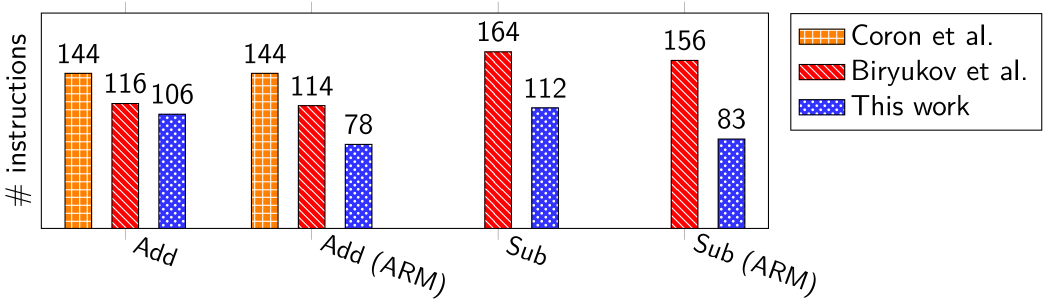
- Combined AND-SHIFT operations save most of the instructions
- Especially when combined with optimizations proposed by Biryukov et el.

# Comparision of masked operations

| | `SecXor` | `SecShift` | `SecAnd` | `SecAndShift` / `-Opt` | `SecAndShiftXor` |
|---|---|---|---|---|---|
| Generic [Coron et al.] | 2 | 4 | 8 | $8+2$ | $8+4+2$ |
| ARM [Coron et al.] | 2 | 4 | 8 | $8+2$ | $8+4+2$ |
| Generic [Biryukov et al.] | 2 | 2 | 7 | $7+2$ | $7+2+2$ |
| ARM [Biryukov et al.] | 2 | 2 | 6 | $6+2$ | $6+2+2$ |
| Generic [new] | 2 | - | 8 | 10 / 9 | 10 |
| ARM [new] | 2 | - | 8 | 8 / 6 | 8 |

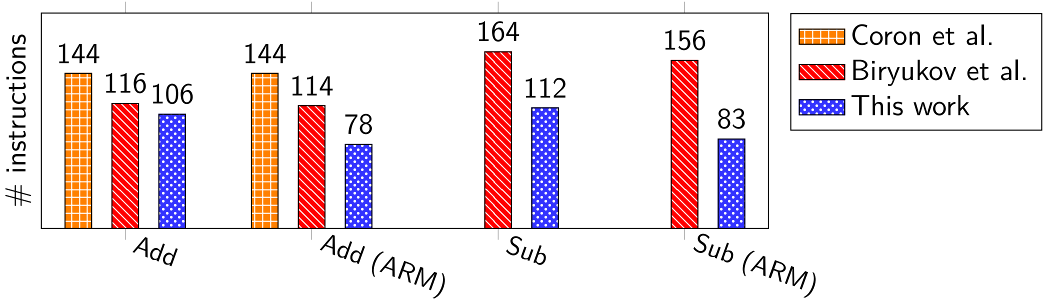
- Combined AND-SHIFT operations save most of the instructions
- Especially when combined with optimizations proposed by Biryukov et el.
- Generation of refresh mask takes only 3 instructions

# Comparision of masked 32-bit modular addition



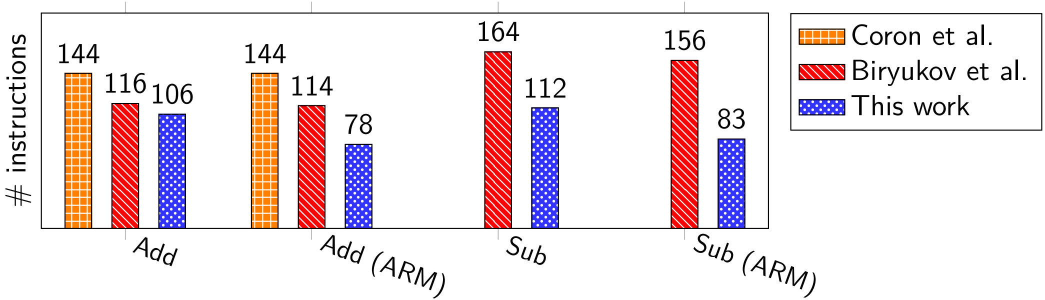
- ARM implementation improved by 31% when combined with approach by Biryukov et al.

# Comparision of masked 32-bit modular addition



- ARM implementation improved by 31% when combined with approach by Biryukov et al.
- Significantly improved subtraction instruction counts

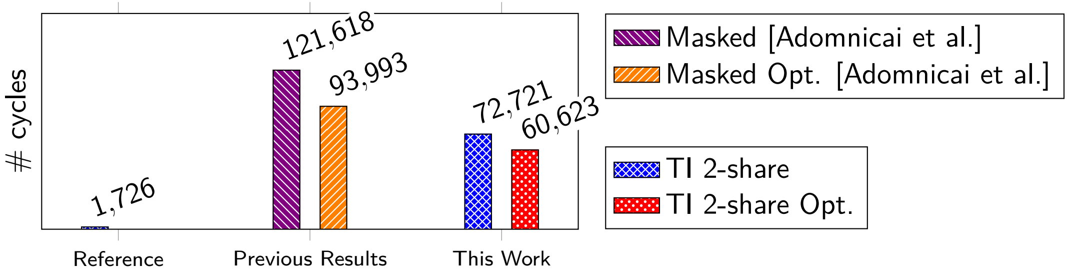
# Comparision of masked 32-bit modular addition



- 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

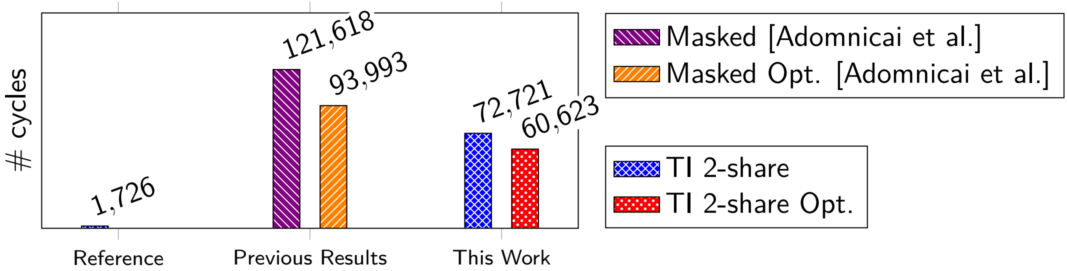
# Application to ChaCha20 cipher

- We implemented an unprotected reference and two protected variants



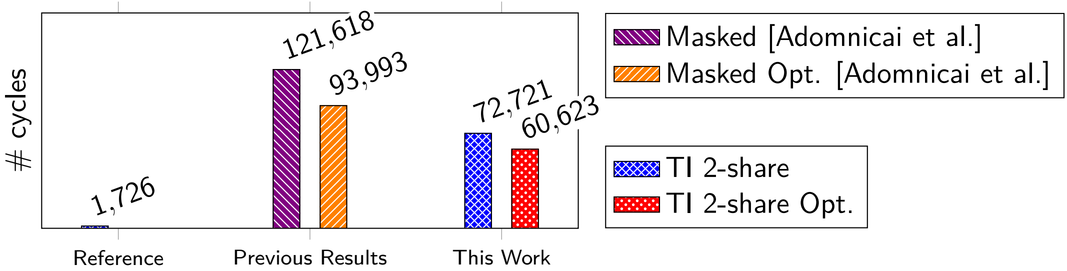
# Application to ChaCha20 cipher

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



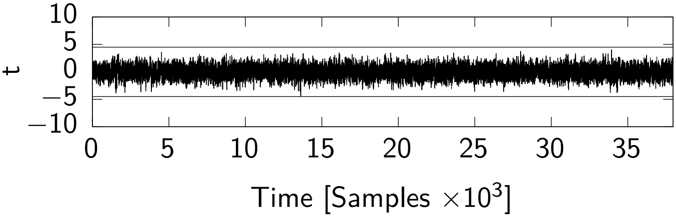
# Application to ChaCha20 cipher

- 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



# 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

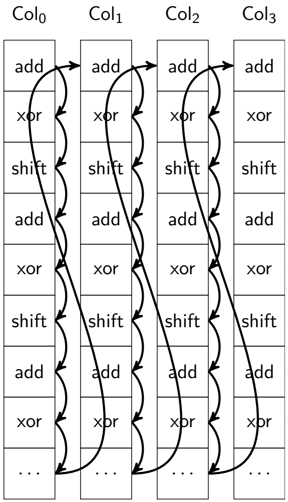


---

[1] https://github.com/cryptolu/maps

Thank you for listening

# Chacha Shuffling (Backup Slide)



- 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
- $\frac{(4 \cdot 12)!}{(12!)^4} \approx 2^{88}$ Permutations
- Noise can be further amplified by splitting the masked addition into several operations

# Chacha Shuffling (Backup Slide)

- 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
- $\frac{(4 \cdot 12)!}{(12!)^4} \approx 2^{88}$ Permutations
- Noise can be further amplified by splitting the masked addition into several operations