Update on Ascon Implementations

Christoph Dobraunig, Maria Eichlseder, Florian Mendel, Martin Schläffer

http://ascon.iaik.tugraz.at
Ascon

Designed in 2014, part of CAESAR lightweight portfolio

• Robust design, lots of security analysis
  • More than 15 publications analyzing the security of Ascon

• Lightweight applications
  • Low cost in HW and SW

• High-performance applications
  • 9 c/b for Ascon-128 and 6 c/b for Ascon-128a in SW
  • High speedup at low cost with lightweight Ascon HW instruction [SP20]

• Balanced design characteristics
  • Close to best in all categories

• Efficient protected implementations
  • Designed with SCA in mind

• Hardware implementations
  • See ASIC and FPGA benchmarking results [KPC20, Moh+20]
Ascon: Low-cost AEAD+Hash

- Ascon-128
- Ascon-128a
- Ascon-Hash
- Ascon-Xof

- same structure
- same permutation
- 320-bit state size
- r={64,128} bit rate
- a=12, b={6,8} rounds

NIST Lightweight Cryptography Workshop 2020
Low-cost components

Non-linear substitution layer

Linear permutation layer

\[
\begin{align*}
    x_0 &= x_0 \oplus (x_0 \gg 19) \oplus (x_0 \gg 28) \\
    x_1 &= x_1 \oplus (x_1 \gg 61) \oplus (x_1 \gg 39) \\
    x_2 &= x_2 \oplus (x_2 \gg 1) \oplus (x_2 \gg 6) \\
    x_3 &= x_3 \oplus (x_3 \gg 10) \oplus (x_3 \gg 17) \\
    x_4 &= x_4 \oplus (x_4 \gg 7) \oplus (x_4 \gg 41)
\end{align*}
\]

parallelizable, use bit-interleaving for 32-bit platforms
Size-optimized implementations

• Low state size: 320 bit state requires only 10 32-bit registers
• Use bit-interleaving for 32-bit (16-bit/8-bit) [Ber+12]
• Small number of temporary registers needed
• Almost no performance overhead for loops
• No table lookups (bitsliced by design, constant time)
• Small overhead of Ascon-128 with Ascon-Hash (+10%)
• Low performance overhead for short messages
• Low-size vs. speed-optimized implementations:
  • 2.2 kB code size with performance penalty 1.42x (32-bit low-end ARM11)
  • 1.7 kB code size with performance penalty 1.09x (64-bit high-end CPU)
  • Crucial for performant protected implementations
Importance of low-cost

• Protect implementations against:
  • SPA, DPA, DFA, SIFA, ...

• Main countermeasures:
  • Masking: factor $x$
  • Hiding: factor $y$
  • Redundancy: factor $z$
  • In practice: factor $x \cdot y \cdot z$

• Importance of:
  • Low overhead for countermeasures
  • Small state/code/area (protected size: $x \cdot y \cdot z$)
Microcontroller benchmarking

\[
\frac{ascon-nocrypt}{best-nocrypt}
\]
for primary submissions @las3

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Uno: 1.34x</td>
<td>• Uno: 3.22x</td>
<td></td>
</tr>
<tr>
<td>• F1: 1.06x</td>
<td>• F1: 1.62x</td>
<td></td>
</tr>
<tr>
<td>• ESP: 1.92x</td>
<td>• ESP: 1.31x</td>
<td></td>
</tr>
<tr>
<td>• F7: 1.02x</td>
<td>• F7: 1.10x</td>
<td>• F7: 1.01x</td>
</tr>
<tr>
<td>• R5: 0.61x</td>
<td>• R5: 1.07x</td>
<td></td>
</tr>
</tbody>
</table>

TBD: use loops for 8-bit implementation

¹https://lwc.las3.de/table.php on 2020/10/14
Ascon: Designed with SCA in mind

• Small state size and code size
• Algebraic degree 2 of S-box
• Efficient masking of S-box
• Limited damage if state is recovered
• Leveled implementations [AFM18, Bel+20]
• Masking using Toffoli gate [Dae+20]

(Some ideas proposed after the design of Ascon but are a perfect fit)
Leveled implementations

- Idea proposed for Ascon in [AFM18]
- Higher protection order for Init/Final (key)
- Lower protection order for AD/PT/CT processing (data)
Masking using Toffoli gate

- Idea proposed in [Dae+20]
- More efficient than masked AND gate
  - Fewer instructions
  - Fewer registers
  - Fewer randomness
- No fresh randomness needed during round computation
  - Randomness is not lost (invertible shared Toffoli gate)
  - Randomness of previous round can be reused
- Benefits of invertible shared function:
  - TI: uniform by design
  - SIFA: reduced attack surface

\[ c \oplus a \& b \]
Masked C code examples

2 shares (init rx.s1 with random)

// 2-share Toffoli gate
#define TOFFOLI(a0, a1, b0, b1, c0, c1) \
    a0 ^= b0 & c0; a0 ^= b0 & c1; \
    a1 ^= b1 & c1; a1 ^= b1 & c0

// 2-share Keccak S-Box of [Dae+20]
rx.s0 = rx.s1;
TOFFOLI(rx.s0, rx.s1, ~x4.s0, x4.s1, x0.s0, x0.s1);
TOFFOLI(x0.s0, x0.s1, ~x1.s0, x1.s1, x2.s0, x2.s1);
TOFFOLI(x2.s0, x2.s1, ~x3.s0, x3.s1, x4.s0, x4.s1);
TOFFOLI(x4.s0, x4.s1, ~x0.s0, x0.s1, x1.s0, x1.s1);
TOFFOLI(x1.s0, x1.s1, ~x2.s0, x2.s1, x3.s0, x3.s1);
x3.s0 ^= rx.s0; x3.s1 ^= rx.s1;

3 shares (init rx.s1, rx.s2 with random)

// 3-share Toffoli gate
#define TOFFOLI(a0, a1, a2, b0, b1, b2, c0, c1, c2) \
     a0 ^= b0 & c0; a0 ^= b0 & c2; a0 ^= b2 & c0; \
    a1 ^= b1 & c1; a1 ^= b1 & c0; a1 ^= b0 & c1; \
    a2 ^= b2 & c2; a2 ^= b2 & c1; a2 ^= b1 & c2

// 3-share Keccak S-Box of [Dae+20]
rx.s2 = rx.s0;
rx.s0 ^= rx.s1;
TOFFOLI(rx.s0, rx.s1, rx.s2, ~x4.s0, x4.s1, x4.s2, x0.s0, x0.s1, x0.s2);
TOFFOLI(x0.s0, x0.s1, x0.s2, ~x1.s0, x1.s1, x1.s2, x2.s0, x2.s1, x2.s2);
TOFFOLI(x2.s0, x2.s1, x2.s2, ~x3.s0, x3.s1, x3.s2, x4.s0, x4.s1, x4.s2);
TOFFOLI(x4.s0, x4.s1, x4.s2, ~x0.s0, x0.s1, x0.s2, x1.s0, x1.s1, x1.s2);
TOFFOLI(x1.s0, x1.s1, x1.s2, ~x2.s0, x2.s1, x2.s2, x3.s0, x3.s1, x3.s2);
x3.s2 ^= rx.s2; x3.s1 ^= rx.s1; x3.s0 ^= rx.s0;
Masked SW performance estimates

• Preliminary C implementation:
  • NIST software interface to benchmark implementations
  • Masks key/data immediately at interface
  • Starting point for dedicated ASM implementations
  • No security guarantee/evaluation of C code

• Performance measurements using:
  • Full masking of Init/Final/AD/PT/CT
  • Straight-forward shared C code using Toffoli gate
  • No additional randomness during round computation

• Hints for ASM implementations:
  • Reduce overhead for register spills
  • Avoid processing shares after each other
  • Process independent even/odd words of bi32 implementation between shares
  • Adapt to leakage characteristics of device
  • Refresh shares on availability of randomness

• High-end performance (64-bit CPU):
  • 2 shares based on opt64
    • Code size: 3.3 kB
    • Performance overhead: 2x
  • 3 shares based on opt64
    • Code size: 6.1 kB
    • Performance overhead: 4x

• Low-end performance (32-bit ARM11):
  • 2 shares based on bi32
    • Code size: 7.4 kB
    • Performance overhead: 3.5x
  • 3 shares based on bi32
    • Code size: 11 kB
    • Performance overhead: 9x

• Independent benchmarking combined with side-channel analysis needed
Summary

• New low-cost Software with limited performance overhead
• C reference code for masking with Toffoli gate
• Designed with SCA in mind, many techniques apply
• Lower overhead for protected implementations
References


