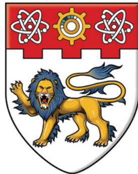
Implementing Lightweight Block Ciphers on x86 Architectures

Jian Guo

joint work with, Ryad Benadjila Victor Lomné Thomas Peyrin



NANYANG
TECHNOLOGICAL
UNIVERSITY

ASK, 27 August 2013
@ Shandong University, Weihai Campus, China

Talk Overview

1. Introduction
2. Table-based
3. Vector-Permutation
4. Bitslice
5. Results and Conclusions

Motivations

Existing work:

- at CHES 2012, Matsuda and Moriai gave the first bitslice implementations on `PRESENT` and `Piccolo`, showing that lightweight block ciphers can perform very well for some cloud applications.
- the good speed assumes the use case where long data is to be enciphered. This may not always be the case, *e.g.*, the Electronic Product Code, being a replacement of barcode, is usually of size 64, 96, 125 bits, under which the speed can be significantly slower.
- also, the key schedule was removed from speed measurement, which does not seem to be a valid assumption for many use cases.

Motivations

Existing work:

- at CHES 2012, Matsuda and Moriai gave the first bitslice implementations on PRESENT and Piccolo, showing that lightweight block ciphers can perform very well for some cloud applications.
- the good speed assumes the use case where long data is to be enciphered. This may not always be the case, *e.g.*, the Electronic Product Code, being a replacement of barcode, is usually of size 64, 96, 125 bits, under which the speed can be significantly slower.
- also, the key schedule was removed from speed measurement, which does not seem to be a valid assumption for many use cases.

Our work:

- consider most of the possible use cases: with short/long data, shared/independent keys, under serial/parallel operation modes.
- besides bitslice, we also apply other implementation techniques, such as table-based, and vector-permutation.
- use LED, Piccolo, and PRESENT as examples.
- give a fair and comprehensive comparison of the speed over all use cases, and over all the three implementation techniques, under test with 6 different devices/servers.

Introduction

Implementation techniques considered:

- Table-Based: table-lookup for sbox implementation, tables need to be prepared in advance; subject to cache-timing attacks [Bernstein 2005].
- Vector-Permutation: introduced by `TWINE` designers (SAC 2012) for better software performance, applies with small parallelism.
- Bitslice: sbox and other components implemented in algebraic forms, good performance comes with computing multiple instances together.

Introduction

Implementation techniques considered:

- Table-Based: table-lookup for sbox implementation, tables need to be prepared in advance; subject to cache-timing attacks [Bernstein 2005].
- Vector-Permutation: introduced by `TWINE` designers (SAC 2012) for better software performance, applies with small parallelism.
- Bitslice: sbox and other components implemented in algebraic forms, good performance comes with computing multiple instances together.

Lightweight block ciphers implemented with all techniques:

- `LED`: 64-bit AES-like design with mainly 64-, 128-bit key size and 32/48 rounds, proposed by Guo *et al*. at CHES 2011.
- `Piccolo`: 64-bit generalized feistel structure with 80-, 128-bit key size and 25/31 rounds, proposed by Shibutani *et al*. at CHES 2011.
- `PRESENT`: 64-bit SP-network design with 80-, 128-bit key size and 31 rounds, proposed by Bogdanov *et al*. at CHES 2007.

Table-based Implementations I

Mainly for designs based on Substition-Permutation Networks, i.e., round function consists of a non-linear operation such as sbox, followed by linear operations, e.g., AES-like designs:

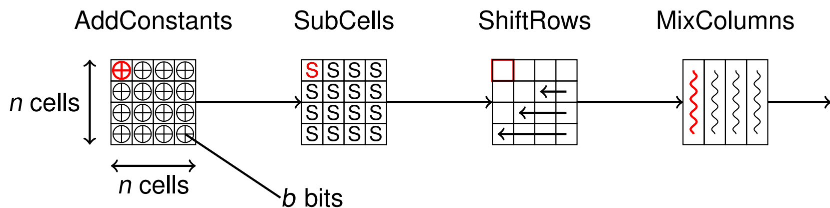


Table-based Implementations II

Implementation Steps:

Preparation: Build tables, with cell input as index, and its corresponding column output as table values.

Table-based Implementations II

Implementation Steps:

Preparation: Build tables, with cell input as index, and its corresponding column output as table values.

Usage:
1. extract the cell value from column/state representation, this involves "shift", and "logic and" operations.
2. table lookups.
3. XOR table lookup values to form round outputs.

Computation of a generic SPN lightweight cipher round

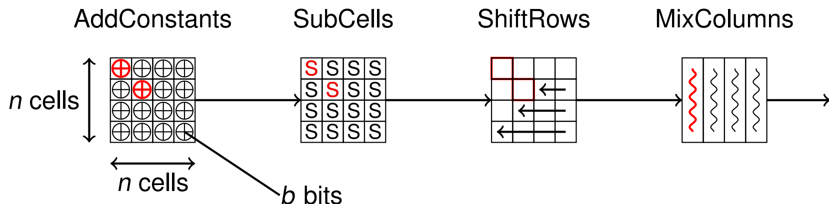
Input: State, Tables / Output: Updated state

```
t0 = T0[ state        & MASKm];
t1 = T1[(state >>  b) & MASKm];
t2 = T2[(state >> 2b) & MASKm];
...
state = t0 ^ t1 ^ t2 ^ ...;
```

Total cost: n^2 times $\gg$, &, **table lookup**, $\oplus$.

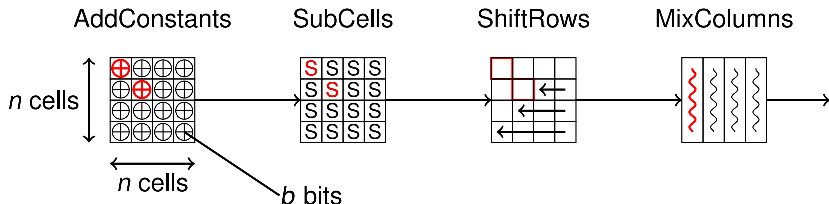
Tabulating

Group m, up to n, cells together to form bigger cells, $1 \leq m \leq n$, then it needs $n \cdot \lceil n/m \rceil$ table-lookups, with bigger memory requirements.
Example with $m = 2$, $n = 4$:



Tabulating

Group m, up to n, cells together to form bigger cells, $1 \leq m \leq n$, then it needs $n \cdot \lceil n/m \rceil$ table-lookups, with bigger memory requirements.
Example with $m = 2$, $n = 4$:



	No. of Tables/Lookups	Memory (bits)	No. of XORs
No Tabulating	n^2	$n^2 \cdot 2^b \cdot nb$	$n \cdot (n-1)$
Tabulating	$\lceil n^2/m \rceil$	$\lceil n^2/m \rceil \cdot 2^{mb} \cdot nb$	$n \cdot (\lceil n/m \rceil - 1)$

Tradeoffs

- memory/table sizes v.s. number of table-lookups, via m. Table size affects speed of lookup operations, due to limitation of cache size.

Tradeoffs

- memory/table sizes v.s. number of table-lookups, via m. Table size affects speed of lookup operations, due to limitation of cache size.
- column v.s. state as lookup table values. Column representation is smaller, while state representation enables integration of other **state-wise operations** such as “ShiftRows”, **inter-column tabulating**, and **SuperSbox technique**.

Tradeoffs

- memory/table sizes v.s. number of table-lookups, via m. Table size affects speed of lookup operations, due to limitation of cache size.
- column v.s. state as lookup table values. Column representation is smaller, while state representation enables integration of other **state-wise operations** such as “ShiftRows”, **inter-column tabulating**, and **SuperSbox technique**.
- SuperSbox for two rounds with more memory requirements v.s. usual table-lookup with less memory requirements for one round.

Deciding the right m

Bigger m implies **more** memory requirements, and **less** table-lookups.
However, if the tables can not be fit into the cache, lookup slows down,

Deciding the right *m*

Bigger *m* implies **more** memory requirements, and **less** table-lookups. However, if the tables can not be fit into the cache, lookup slows down, by **how much** ?

Deciding the right m

Bigger m implies **more** memory requirements, and **less** table-lookups. However, if the tables can not be fit into the cache, lookup slows down, by **how much** ?

microarchitecture	$L1$ **size** (KBytes)	$L1$ **latency** (cycles)	$L2$ **size** (KBytes)	$L2$ **latency** (cycles)
Intel *P6*	16 or 32	3	512	8
Intel Core	32	3	1500	15
Intel Nehalem / Westmere	32	4	256	10
Intel Sandy / Ivy Bridge	32	5	256	12

Deciding the right m

Bigger m implies **more** memory requirements, and **less** table-lookups. However, if the tables can not be fit into the cache, lookup slows down, by **how much** ?

microarchitecture	$L1$ **size** (KBytes)	$L1$ **latency** (cycles)	$L2$ **size** (KBytes)	$L2$ **latency** (cycles)
Intel *P6*	16 or 32	3	512	8
Intel Core	32	3	1500	15
Intel Nehalem / Westmere	32	4	256	10
Intel Sandy / Ivy Bridge	32	5	256	12

$$l_T = P_{L1} \times l_{L1} + P_{L2} \times l_{L2} + P_{L3} \times l_{L3} + P_M \times l_M + \cdots$$

So that we can "predict" the best choice of m, without actual implementations.

Deciding the right *m*

Bigger *m* implies **more** memory requirements, and **less** table-lookups. However, if the tables can not be fit into the cache, lookup slows down, by **how much** ?

microarchitecture	$L1$ **size** (KBytes)	$L1$ **latency** (cycles)	$L2$ **size** (KBytes)	$L2$ **latency** (cycles)
Intel *P6*	16 or 32	3	512	8
Intel Core	32	3	1500	15
Intel Nehalem / Westmere	32	4	256	10
Intel Sandy / Ivy Bridge	32	5	256	12

$$l_T = P_{L1} \times l_{L1} + P_{L2} \times l_{L2} + P_{L3} \times l_{L3} + P_M \times l_M + \cdots$$

So that we can "predict" the best choice of *m*, without actual implementations.

Observations: for better performance, feed $L1$ cache as much as possible, and in most of the cases, exceeding a bit the $L1$ cache is better than partial-usage, *e.g.*, $m = 2$ gives the best speed for LED, and it is faster when $m = 3$ than that when $m = 1$.

Super-Sbox Technique for AES-like Designs

$AC \circ SB \circ SR \circ MC \circ AC \circ SB \circ SR \circ MC$

$\Longleftrightarrow$

$AC \circ SR | \underbrace{SB \circ MC \circ AC \circ SB}_{\text{supersbox}} | SR \circ MC$

Super-Sbox Technique for AES-like Designs

$AC \circ SB \circ SR \circ MC \circ AC \circ SB \circ SR \circ MC$

$\Longleftrightarrow$

$AC \circ SR | \underbrace{SB \circ MC \circ AC \circ SB}_{\text{supersbox}} | SR \circ MC$

Cipher can be viewed as:

$$AC \circ SR \cdots | \underbrace{SB \circ MC \circ AC \circ SB \quad SR \circ MC \circ AC \circ SR}_{repeat\ \ r/2\ \ times} | \cdots \ SR^{-1} \circ AC^{-1}$$

Super-Sbox Technique for AES-like Designs

$AC \circ SB \circ SR \circ MC \circ AC \circ SB \circ SR \circ MC$

$\Longleftrightarrow$

$AC \circ SR | \underbrace{SB \circ MC \circ AC \circ SB}_{\text{supersbox}} | SR \circ MC$

Cipher can be viewed as:

$$AC \circ SR \cdots | \underbrace{SB \circ MC \circ AC \circ SB \quad SR \circ MC \circ AC \circ SR}_{repeat\ \ r/2\ \ times} | \cdots \ SR^{-1} \circ AC^{-1}$$

$SB \circ MC \circ AC \circ SB \circ SR \circ MC \circ AC \circ SR$ forms the new SP-Network, with

- $(SB \circ MC \circ AC \circ SB)$ as the S-layer, with column as new sbox.
- $(SR \circ MC \circ AC \circ SR)$ as the P-Layer

Super-Sbox Technique for AES-like Designs

$AC \circ SB \circ SR \circ MC \circ AC \circ SB \circ SR \circ MC$

$\Longleftrightarrow$

$AC \circ SR | \underbrace{SB \circ MC \circ AC \circ SB}_{\text{supersbox}} | SR \circ MC$

Cipher can be viewed as:

$$AC \circ SR \cdots | \underbrace{SB \circ MC \circ AC \circ SB \quad SR \circ MC \circ AC \circ SR}_{repeat\ \ r/2\ \ times} | \cdots \ SR^{-1} \circ AC^{-1}$$

$SB \circ MC \circ AC \circ SB \circ SR \circ MC \circ AC \circ SR$ forms the new SP-Network, with

- ($SB \circ MC \circ AC \circ SB$) as the S-layer, with column as new sbox.
- ($SR \circ MC \circ AC \circ SR$) as the P-Layer

Effects:

Cons: number of entries for each table increases to 2^{nb}, with state as table values, this requires $2^{nb} \cdot n^2 b \cdot n \cdot (r/2 - 1)$ bits memory, e.g., LED-64 requires 32 MB.

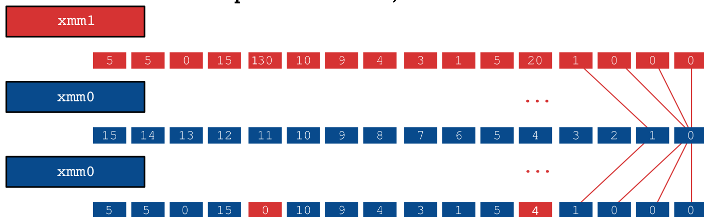
Pros: Reduce the number of rounds by half. On some processors and likely on future bulldozer processors, it might improve the speed.

Vector-Permutation

`vperm` for short, was introduced by `TWINE` designers in SAC 2012, takes advantage of SIMD (single instruction multiple data), especially “pshufb” instruction, introduced with SSSE3 extension with Intel Core microarchitecture.

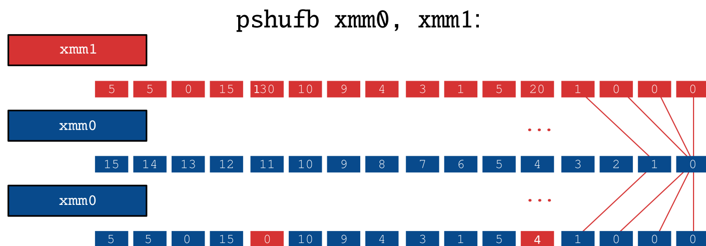
Vector-Permutation

vperm for short, was introduced by TWINE designers in SAC 2012, takes advantage of SIMD (single instruction multiple data), especially “pshufb” instruction, introduced with SSSE3 extension with Intel Core microarchitecture.



Vector-Permutation

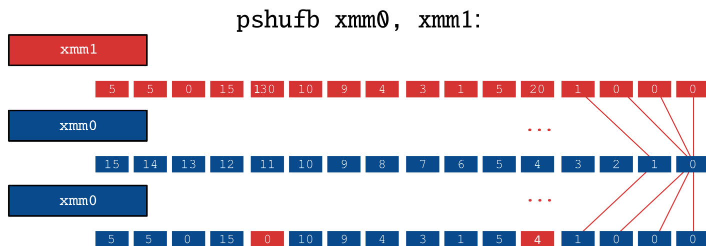
vperm for short, was introduced by TWINE designers in SAC 2012, takes advantage of SIMD (single instruction multiple data), especially "pshufb" instruction, introduced with SSSE3 extension with Intel Core microarchitecture.



When xmm0 stores $s[0], s[1], \ldots$, and sbox inputs are stored in xmm1 byte-wise, then this instruction functions as substitution byte-wise, in parallel. Perfectly fits 4-bit sboxes, with possibility extending to larger outputs.

Vector-Permutation

vperm for short, was introduced by TWINE designers in SAC 2012, takes advantage of SIMD (single instruction multiple data), especially “pshufb” instruction, introduced with SSSE3 extension with Intel Core microarchitecture.

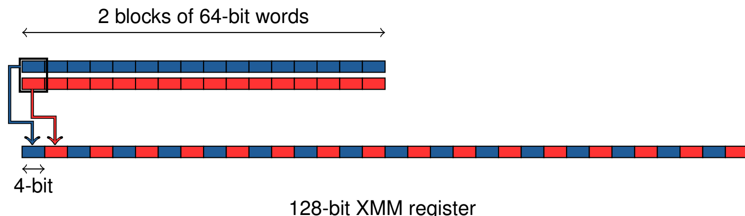


When xmm0 stores $s[0], s[1], \ldots$, and sbox inputs are stored in xmm1 byte-wise, then this instruction functions as substitution byte-wise, in parallel. Perfectly fits 4-bit sboxes, with possibility extending to larger outputs.

Besides simple design TWINE, it turns out to be efficient for relatively more complex designs as well.

Packing and unPacking

Packing for the 2-parallelism `vperm` implementation of `PRESENT` and `Piccolo`. Each rectangle represents a 4-bit nibble:



Substitution Operation for 2-parallel vperm

```
x =       s & 0x0f0f0f...0f; vpshufb r0, t0, x;
y = (s>>4) & 0x0f0f0f...0f; vpshufb r1, t1, y;
r = r0 ^ r1
```

Note: $t1 = t0 \ll 4$.

Development and Future

- New instruction sets are added to SSSE4.2, available on Intel CPU generations, Sandy and Ivy Bridge. 128-bit `xmm` registers are extended to 256-bit `ymm` registers.
- Three operands form of the instructions, table lookups can be performed with one instruction "`vpshufb t, s, r`" instead of the two instuctions "`movdqa t, s; pshufb t, r`"
- full operations on `ymm` have been introduced on the recent Haswell architecture with `AVX2`, to be extended to 512-bit registers in around 2015.

Use cases

Motivated by the use case of encryption/decryption with big data under same key considered in [Matsuda-Moriai'12], and to fit applications with short data and distinct keys such as authentication and access control, we further generalize the use cases according to three factors:

- the server is communicating with **D** devices, each using a distinct key
- the server has to encrypt/decrypt **B** blocks of data, each block of 64 bits.
- the operation modes, **serial** (such as CBC) and **parallel** (such as CTR).

These factors are used to test table-based and `vperm` implementations as well.

Bitslice Implementations

- sboxes are represented and implemented in algebraic forms
- usually runs with multiple instances of data together, for better performance
- all cipher components need to be implemented in bitslice way, including key schedules if any
- come with overhead of packing and unpacking of both data and key materials into bitslice form, which in some cases can cause performance reduction significantly

Algebraic Forms of Sboxes

Bitslice implementation of LED and PRESENT Sbox

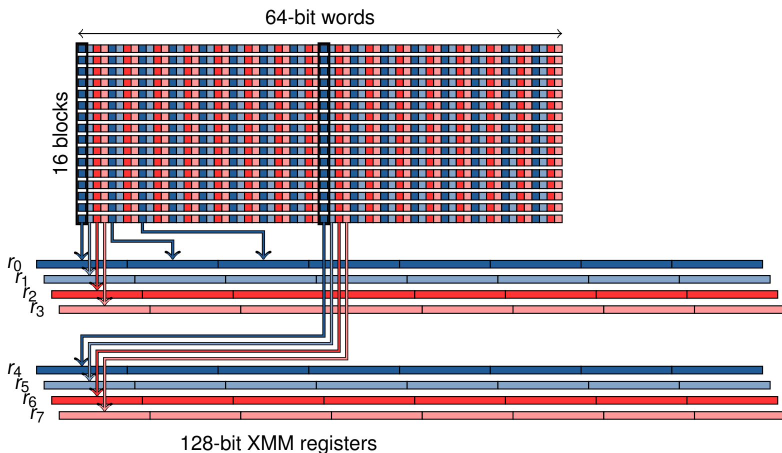
```
// Input: r3, r2, r1, r0, t  ---  Output: r3, r2, r1, r0
#define Sbox(r3, r2, r1, r0, t)
r2 = XOR(r2,r1);    r3 = XOR(r3,r1);    t  = r2;          r2 = AND(r2,r3);
r1 = XOR(r1,r2);    t  = XOR(t,r0);     r2 = r1;          r1 = AND(r1,t);
r1 = XOR(r1,r3);    t  = XOR(t,r0);     t  = OR(t,r2);    r2 = XOR(r2,r0);
r2 = XOR(r2,r1);    t  = XOR(t,r3);     r2 = ~r2;         r0 = XOR(r0,t);
r3 = r2;            r2 = AND(r2,r1);    r2 = XOR(r2,t);   r2 = ~r2;
```

Bitslice implementation of Piccolo Sbox

```
// Input: r3, r2, r1, r0, t  ---  Output: r0, r1, r2, r3
#define Sbox(r3, r2, r1, r0, t)
t  = r1;            r1 = OR(r1,r2);     r3 = ~r3;           r0 = XOR(r0,r2);
r1 = XOR(r1,r3);    r3 = OR(r3,r2);     r0 = XOR(r0,r3);    r3 = r1;
r3 = OR(r3,r0);     r3 = XOR(r3,t);     t  = OR(t,r0);      r2 = XOR(r2,t);
r3 = ~r3;
```

There can be many algebraic forms for the same sbox, (sub-) optimal choice search is mainly done by the tool provided by Dag Arne Osvik in 2000.

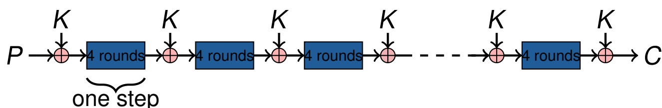
Packing and Unpacking



Packing for the 16-parallel bitslice implementation of `LED` and `PRESENT`.

Key Schedules

- LED has virtually no key schedule, the same master key is used as round key in every 4 rounds.



- PRESENT key schedule consists of sbox, adding constant, and rotation of 61 bits;
- Piccolo key schedule involves selection and combination of 16-bit key words, we prepare the bitslice form of the 16-bit words independently once and for all, and combination can be done with xor-ing the words at each round.

	LED-64	LED-128	Piccolo-80	Piccolo-128	PRESENT-80	PRESENT-128
Key schedule ratio	3.3%	4.1%	20.2%	26.7%	55.2%	59.9%

Use cases and Tests

	indep. devices	mess. size	op. mode	example	LED	PRESENT	Piccolo
1	small	small	-	authentication / access control / secure traceability (industrial assembly line)	tbl/vperm	tbl/vperm	tbl/vperm
2	small	big	parallel	secure streaming communication (medical device sending continuously sensitive data to a server, tracking data, etc.)	bitslice	bitslice	bitslice
3	small	big	serial	secure serial communication	tbl/vperm	tbl/vperm	tbl/vperm
4	big	small	-	multi-user authentication / secure traceability (parallel industrial assembly lines)	bitslice	bitslice	bitslice
5	big	big	parallel	multi-user secure streaming communication / cloud computing / smart meters server / sensors network / Internet of Things	bitslice	bitslice	bitslice
6	big	big	serial	multi-user secure serial communication	bitslice	bitslice	bitslice

Six device/server use cases for lightweight encryption. The notation “big/small” refers to more/less than 10 on average, for experiments we used 1000 and 1.

Our Results

- implemented each technique to all main variants of `LED`, `Piccolo`, `PRESENT`, and tested under different use cases, with 6 devices/servers.

Our Results

- implemented each technique to all main variants of `LED`, `Piccolo`, `PRESENT`, and tested under different use cases, with 6 devices/servers.
- provided a model to predict table-based implementation speed for best choice of tabulating.

Our Results

- implemented each technique to all main variants of `LED`, `Piccolo`, `PRESENT`, and tested under different use cases, with 6 devices/servers.
- provided a model to predict table-based implementation speed for best choice of tabulating.
- observed the key schedule, and (un)packing overhead in bitslice implementations.

Our Results

- implemented each technique to all main variants of LED, Piccolo, PRESENT, and tested under different use cases, with 6 devices/servers.
- provided a model to predict table-based implementation speed for best choice of tabulating.
- observed the key schedule, and (un)packing overhead in bitslice implementations.
- achieved best LED table-based implementation of 57c/B, and best bitslice implementation of 12 c/B.

Our Results

- implemented each technique to all main variants of LED, Piccolo, PRESENT, and tested under different use cases, with 6 devices/servers.
- provided a model to predict table-based implementation speed for best choice of tabulating.
- observed the key schedule, and (un)packing overhead in bitslice implementations.
- achieved best LED table-based implementation of 57c/B, and best bitslice implementation of 12 c/B.
- overview of the way to choose the best implementation technique for different use cases.

Our Results

- implemented each technique to all main variants of `LED`, `Piccolo`, `PRESENT`, and tested under different use cases, with 6 devices/servers.
- provided a model to predict table-based implementation speed for best choice of tabulating.
- observed the key schedule, and (un)packing overhead in bitslice implementations.
- achieved best `LED` table-based implementation of 57c/B, and best bitslice implementation of 12 c/B.
- overview of the way to choose the best implementation technique for different use cases.
- confirmed that lightweight block ciphers can be efficient, with at least one of the implementation techniques.

Our Results

- implemented each technique to all main variants of LED, Piccolo, PRESENT, and tested under different use cases, with 6 devices/servers.
- provided a model to predict table-based implementation speed for best choice of tabulating.
- observed the key schedule, and (un)packing overhead in bitslice implementations.
- achieved best LED table-based implementation of 57c/B, and best bitslice implementation of 12 c/B.
- overview of the way to choose the best implementation technique for different use cases.
- confirmed that lightweight block ciphers can be efficient, with at least one of the implementation techniques.
- all source code will be published via https://sites.google.com/site/cipherscodes

Thank you !

Q & A