6.2. Implementation

Figure 6.3: Bitslice setup of consecutive blocks

Equation 6.1.

\[
f(i) = \begin{cases} 
32 & i \leq r \\
32 - (i - r) & r < i \leq r + 32 \\
0 & r + 32 < i 
\end{cases}
\]  

We propose a bitslice implementation with input data from different rounds, which is possible although operations involving unknown bits must be repeated when the input data is available. The capacity of an \(N\)-bit processor will be fully exploited in Boolean operations with an operand bit index lower than \(32 - N + 1\).

For the proposal explanation, we consider that the algorithm performs the update shift operation to the right. This is the case of the KeeLoq encryption algorithm. If the update shift is performed to the left, as occurs in KeeLoq decryption, the references to LSB would change to MSB.

In FSR-based algorithms, the \(i\)-th bit in a round is the \((i-1)\)-th bit in the next round, excepting the LSB. The LSB of the \(i\)-th round is the \(i\)-th bit of the first round. Input data is already setup for the bitslice implementation between rounds, reducing the setup overhead. The methodology we propose is to use an ALU-width variable for each input of the feedback function. The content of each variable is obtained by shifting the input data word until the corresponding bit is the LSB. The content of the \(i\)-th bit variable results from shifting the input data \(i\) times. The variable for bit 0 is the lower word of input data. For a given \(j\)-bit input data and an \(n\)-bit ALU, the amount of valid bits in a \(i\)-th bit variable is \(\min(j - i, n)\).

We propose a bitslice implementation using nested loops. For an \(N\)-width CPU, the external loop represents \(N\) rounds. As KeeLoq requires 528 rounds, this loop will be iterated 528/\(N\) times, following the structure mentioned at the beginning of this Chapter.

Operations with two operands with \(N\) valid bits are performed once for \(N\) rounds. The rest are performed in an inner loop as valid bits are available.
For $N$ greater than 4, an inner loop would include operations with more than 1-bit known-data but less than $N$. Depending on the bits available and the number of operations, the inner loop can be implemented with $N/4$ or $N/2$ iterations.

We develop a test case with KeeLoq and MSP430, with 16-bit ALU. In KeeLoq encryption algorithm, operations involving bits $x_{20}$ (if $N$ is 16) and $x_{26}$ would fit in the inner loop, which performs operations of 4 rounds per iteration ($N/4$) with 4 iterations. In KeeLoq decryption algorithm, operations involving $x_8$ would fit in the inner loop performing operations from 8 rounds per iteration ($N/2$) with 2 iterations.

Finally, an inner loop with operations of 1 round per iteration would include operations involving the output bit of the previous iterations, which is $x_{31}$ for encryption and $x_0$ for decryption. This loop would iterate the amount of rounds performed in the immediately upper loop, which is 4 times for encryption and 8 rounds for decryption.

A KeeLoq encryption implementation of the external loop content for 16-bit processor is shown in 6.2.

**Listing 6.2: Bitslice implementation**

```c
size_t d0, d1, d9, d16, d20, d26, d31, d32;
size_t dd, d1_9, d20_26, mask_calc, tmp, tmp_in, j, l;
...
d0 = data.i16.low;
d1 = data.i32 >> 1;
d9 = data.i32 >> 9;
d16 = data.i16.high;
d1_9 = d1 ^ d1&d9;
dd = key ^ d0 ^ d16 ^ d9 ^ d1_9;
d20 = data.i16.high >> 4;
d26 = data.i16.high >> 10;
d31 = data.i16.high >>15;
d32 = 0;
mask_calc = 1;
for (j = 4; j > 0; j--) {
d20_26 = d26&d20;
tmp = dd ^ d20_26 ^ d26&d1 ^ d20&d9;
tmp_in = d1_9 ^ d20 ^ d20&d1 ^ d26&d9 ^ d20_26;
for (l = 4; l > 0; l--) {
d32 |= (tmp_in&d31 ^ tmp) & mask_calc;
d31 |= (d32 << 1);
mask_calc <<= 1;
}
d26 |= (d32 * 64);
d20 |= (d32 * 4096);
}

//Update data
data.i16.low = data.i16.high;
data.i16.high = d32;
```

The bitslice implementation has divided the execution time by 4, compared to the basic ANF-based implementation that is used as source for the optimization. The potential
6.2. Implementation

<table>
<thead>
<tr>
<th></th>
<th>exec cycles</th>
<th>Code size</th>
<th>Register usage</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>max</td>
<td>(bytes)</td>
<td>static</td>
</tr>
<tr>
<td>bitslice</td>
<td>14189</td>
<td>14189</td>
<td>408</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 6.6: Performance evaluation of bitslice implementation

<table>
<thead>
<tr>
<th>bit</th>
<th>bit 31</th>
<th>bit 26</th>
<th>bit 20</th>
<th>bit 9</th>
<th>bit 1</th>
<th>bit 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitslice</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.7: Extra cycles when input bit active of bitslice

An improvement limit would be 16 (16 parallel Boolean units in bitslice). However, not every input data is available for performing 16 operations at a time and there is an overhead in the parallelization process.

The overhead in data memory is negligible, only 4 more bytes. The overhead in code size is 80 bytes, a 24% of overhead. Compared to the gen_tb041, the fastest LUT-based implementation, it improves the execution time by a factor 2.24, with a code size overhead of 2.17 and using less data memory. Compared to the tb041 and nlf_tb041, the execution time improvement factor is greater than 3 with a code size overhead less than 2, and only 18 more bytes of data memory (a 24% of increase).

Considering security, this implementation completely meets condition 2, as it has an execution path independent of input or key data. Table 6.7 is a summary of the deviations related to input data, which is always zero. Moreover, it is not achieved by balancing different execution branches, but by having only one execution branch.

We expect that the lack of timing deviation and data dependent branches makes the CPA more efficient, as the intermediate value is processed always at the same time instant.

Figure 6.4b shows that the correlation value and the number of measurements required for a successful attack is lower than for LUT-based implementations.

It is mandatory to introduce carefully randomness in amplitude, time or both of possible intermediate values to increase the resistance against differential SCA, condition 3 for a good implementation, avoiding major effects on performance and timing leakage. We propose the insertion of dummy rounds, as done in [102]. Each round of the external loop of the proposed bitslice implementation performs $N$ rounds. However, we could use $0, N/4, N/2$ or $3N/4$, wasting the discarded output values of the round.

The bitslice implementation uses an update function that discards the lower word of the input data, moves the upper word to become the lower word of the next iteration and inserts the calculated new data as the upper word. The concrete implementation is done in two C instructions, as shown in bitslice code in Listing 6.2. We propose to substitute the update data function used between external loop iterations with a function that is randomly selected between five possible candidates. Depending on the selected function, part of the calculated data is discarded. Figure 6.5.a depicts the process while Figure 6.5.b shows the optimal algorithm to perform the new 5 update functions.
Chapter 6. Countermeasure proposal III: bit-level parallelism optimization

(a) CPA result

(b) Resistance against CPA

Figure 6.4: CPA of bitslice implementation

<table>
<thead>
<tr>
<th>Input data</th>
<th>Output data</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3</td>
<td>L2</td>
</tr>
</tbody>
</table>

**Next input data (rounds selected)**

<table>
<thead>
<tr>
<th>0</th>
<th>N/4</th>
<th>N/2</th>
<th>3N/4</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3</td>
<td>L2</td>
<td>L1</td>
<td>L0</td>
<td>H0</td>
</tr>
</tbody>
</table>

**a) data update**

- mov low:
  - low << 4
  - low & 0x00F0
  - swap low
  - high >> 4
  - high | low

- high & 0x0FF
  - low & 0x00F0
  - swap low
  - swap high
  - high | low

**b) data update optimal implementation**

<table>
<thead>
<tr>
<th>rra high</th>
<th>rra low</th>
<th>rra low</th>
<th>rra low</th>
<th>rra low</th>
<th>rra low</th>
<th>rra low</th>
<th>rra low</th>
<th>rra low</th>
</tr>
</thead>
<tbody>
<tr>
<td>rra high</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
</tr>
<tr>
<td>rra high</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
</tr>
<tr>
<td>rra high</td>
<td>rla high</td>
<td>rra low</td>
<td>rra low</td>
<td>rla high</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
</tr>
<tr>
<td>rra high</td>
<td>rla high</td>
<td>rra low</td>
<td>rra low</td>
<td>rla high</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
</tr>
<tr>
<td>rra high</td>
<td>rla high</td>
<td>rra low</td>
<td>rra low</td>
<td>rla high</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
</tr>
<tr>
<td>rra high</td>
<td>rla high</td>
<td>rra low</td>
<td>rra low</td>
<td>rla high</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
<td>rra low</td>
</tr>
<tr>
<td>swpb high</td>
<td>swpb high</td>
<td>swpb high</td>
<td>swpb low</td>
<td>swpb low</td>
<td>swpb low</td>
<td>swpb low</td>
<td>swpb low</td>
<td>swpb low</td>
</tr>
<tr>
<td>and high, 0x00FF</td>
<td>and high, 0xFF00</td>
<td>and high, 0xFF00</td>
<td>and high, 0x00FF</td>
<td>and high, 0xFF00</td>
<td>and high, 0x00FF</td>
<td>and high, 0xFF00</td>
<td>and high, 0x00FF</td>
<td>and high, 0xFF00</td>
</tr>
<tr>
<td>bis high, low</td>
<td>bis high, low</td>
<td>bis high, low</td>
<td>bis high, low</td>
<td>bis high, low</td>
<td>bis high, low</td>
<td>bis high, low</td>
<td>bis high, low</td>
<td>bis high, low</td>
</tr>
</tbody>
</table>

**c) SCARE-resistant data update**

Figure 6.5: Update data
Discarding part of the calculated data is equivalent to inserting dummy rounds: it inserts timing deviations that are not data dependent. We expect the efficiency of differential SCA to be reduced. However, the effect of inserting random dummy rounds was mitigated in [102] because the attackers could distinguish one valid round from a dummy round. It is mandatory to implement identically every possible branch in our proposal. The only difference between different rounds is the update function. An implementation based on the optimal algorithm shown in Figure 6.5.b would lead to timing information leakage regarding the number of valid rounds. Moreover, a simple timing aware implementation that adds NOP instructions would not be enough. In [66] the authors present SCA Reverse Engineering (SCARE), a methodology to distinguish the instruction sequence executed from power traces. The implementation must perform the same instruction sequence, with different operands. Figure 6.5.c shows the implementation we propose, resistant to SCARE, for the 5 update functions in a 16-bit processor. With this solution we introduce timing randomness, as the attacker can not synchronize the traces with an intermediate value.

Amplitude randomness is introduced exploiting the wasted operations of discarded rounds. Rounds wasting valid data are performing valid operations, although the output is withdrawn. We propose to modify randomly the discarded bits, with an XOR operation. The noise introduced modifies the information leaked by intermediate values. Similar to random dummy instruction countermeasures, a random number generator is required to execute the countermeasures correctly.

The potential overhead in execution time of the sec-bitslice implementation compared to bitslice duplicates execution time, if round selection is uniformly distributed. However, the new update function also introduces execution overhead, turning in a 2.47 mean overhead (a mean of 35127 cycles). The executing time of the fastest execution, selecting always the update function with no discards ($N$), is 17604 cycles. The overhead in execution time due to the new update function is 24%. The code size is incremented by a factor of 2.68. The extra code includes the 5 different update functions and the random manipulation of bits to be discarded.

Compared to $tb041$ implementation it still reduces the execution time by a factor of at least 1.21, and it has similar execution time than gen_tb041.

Figure 6.6a depicts the result of CPA on the sec-bitslice proposal. The result is similar to CPA on bitslice (Figure 6.4a). However, the correlation coefficient is lower and the top value does not correspond to the correct key guess. Figure 6.6b shows the resistance against CPA of the sec-bitslice on the first iteration of external loop. The maximum number of samples used to measure the resistance is 50000, an order of magnitude greater than with the other implementations, and the attack is still not successful.

<table>
<thead>
<tr>
<th></th>
<th>exec cycles</th>
<th>Code size</th>
<th>Register usage</th>
<th>Memory usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>max</td>
<td>(bytes)</td>
<td>static</td>
</tr>
<tr>
<td>sec-bitslice</td>
<td>33531</td>
<td>36723</td>
<td>1094</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 6.8: Performance evaluation of secured bitslice implementation
Chapter 6. Countermeasure proposal III: bit-level parallelism optimization

(a) CPA result

![CPA result](image)

(b) Resistance against CPA

Figure 6.6: CPA of sec-bitslice implementation

<table>
<thead>
<tr>
<th>Implementation</th>
<th>exec cycles</th>
<th>Code size</th>
<th>Register usage</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>max</td>
<td>(bytes)</td>
<td>static</td>
</tr>
<tr>
<td>tb041</td>
<td>42850</td>
<td>43849</td>
<td>244</td>
<td>11</td>
</tr>
<tr>
<td>nlftb041</td>
<td>43206</td>
<td>44470</td>
<td>252</td>
<td>11</td>
</tr>
<tr>
<td>gentb041</td>
<td>31850</td>
<td>32311</td>
<td>188</td>
<td>10</td>
</tr>
<tr>
<td>anf</td>
<td>57411</td>
<td>57482</td>
<td>328</td>
<td>12</td>
</tr>
<tr>
<td>bitslice</td>
<td>14189</td>
<td>14189</td>
<td>408</td>
<td>12</td>
</tr>
<tr>
<td>sec-bitslice</td>
<td>33531</td>
<td>36723</td>
<td>1094</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 6.9: Performance evaluation of KeeLoq implementations

Considering the 4 different windows of samples plotted in Figure 6.6a with the vertical lines, the attack with 50000 samples is still not successful.

6.3 Results

In this chapter we have evaluated three characteristics of the known LUT-based implementations: performance, timing analysis and differential SCA. We have proposed three new implementations based on the ANF of the KeeLoq algorithm, that have been evaluated. Table 6.9 summarizes the performance results of the six evaluated implementations. The amount of variables used in the different evaluations is very different. Figure 6.7 depicts the amount of data memory accesses of the implementations, distinguishing between outer and inner loop iteration.

Considering the timing analysis, the ANF-based do not leak information, specially the bitslice implementations. Table 6.10 sums up the timing deviation due to key and input data.

Table 6.11 shows a summary of the resistance of these implementations against CPA.
Figure 6.7: Data memory access in the inner and outer loops

<table>
<thead>
<tr>
<th></th>
<th>bit 31</th>
<th>bit 26</th>
<th>bit 20</th>
<th>bit 9</th>
<th>bit 1</th>
<th>bit32</th>
</tr>
</thead>
<tbody>
<tr>
<td>tb041</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>nlftb041</td>
<td>0</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>gentb041</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>anf</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>bitslice</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sec-bitslice</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.10: Extra cycles when input bit active of LUT-based implementations
### Chapter 6. Countermeasure proposal III: bit-level parallelism optimization

#### Table 6.11: Summary of resistance against CPA of KeeLoq implementations

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Maximum Min. samples</th>
<th>Stable</th>
<th>More than 5% over max Min. samples</th>
<th>Stable</th>
<th>(\rho) @ 2000 samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>tb041</td>
<td>43</td>
<td>710</td>
<td>79</td>
<td>1850</td>
<td>0.329175</td>
</tr>
<tr>
<td>nlftb041</td>
<td>117</td>
<td>136</td>
<td>440</td>
<td>810</td>
<td>0.449899</td>
</tr>
<tr>
<td>gentb041</td>
<td>23</td>
<td>91</td>
<td>103</td>
<td>153</td>
<td>0.666491</td>
</tr>
<tr>
<td>anf</td>
<td>14</td>
<td>58</td>
<td>15</td>
<td>144</td>
<td>0.767303</td>
</tr>
<tr>
<td>bitslice</td>
<td>17</td>
<td>19</td>
<td>77</td>
<td>77</td>
<td>0.967898</td>
</tr>
<tr>
<td>sec-bitslice</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.825810</td>
</tr>
</tbody>
</table>

In order to consider the key to be extracted from Figures 6.1.b, 6.1.d, 6.2a, 6.4.b and 6.6.b we consider two possible conditions: first, the correct key guess is the one with greater correlation factor; second, the correlation factor of the correct key guess is more than 5% over the rest of correlation factors. In both cases, we consider the minimum amount of samples to achieve the threshold and the minimum amount of samples from which the condition is achieved with stability (for 10 consecutive experiments).

The non-secured ANF-based implementations are weaker to CPA than LUT-based implementations. The proposed secured bitslice implementation of KeeLoq is secure to CPA, as a CPA using 50000 samples on a clean and aligned implementation is not successful.

### 6.4 Automation

The bitslice implementation can be obtained automatically when the algorithm is described using Boolean operations (with the ANF expression of the NLF). It can be obtained from the programmer or automatically, from the NLF definition.

We extend the LLVM compiler with an intrinsic function for NLF. An intrinsic function is converted by the compiler to concrete instructions with knowledge about the optimizations applied and the concrete target.

We propose the naming convention NLF_<LUT width>_<status width> for declaring NLF intrinsics. For KeeLoq, we define an intrinsic called NLF_32_32, as the NLF is defined by 5 inputs (a LUT word of 32-bits) taken from a 32-bit status word. Intrinsics for other combinations would follow the same naming convention.

The default implementation for the intrinsic is similar to the tb041. The inputs form an index to select a value from the LUT word. An optimization pass transforms the intrinsic into the ANF-based implementation, if loaded. The algorithm for the ANF-based implementation is divided in two stages: “candidate exploration” and “candidate selection”.

The “candidate exploration” stage generates an ordered list of every possible XOR operand of the ANF. Every operand is an AND operation of the input bits. For KeeLoq, the candidates are the 32 resulting combinations of the 5 inputs. The naming convention
for the candidates are a mask of the operands involved in the AND operation, being \texttt{00000}
the no operand AND operation (1) and \texttt{11111} an AND operation of all the inputs. The
output of this stage has always \(2^n\) elements, being \(n\) the number of inputs of the inputs.
The list is ordered increasingly according to the HW of the elements; i.e., the operations
with less operands involved in the AND operation will be considered first. For similar
HW, the lower bits are considered first. The first candidate is always the no operand
candidate, followed by \(n\) 1-operand candidates (no AND operation).

Listing 6.3: candidate exploration implementation

```c
buffer[new++] = 0;
for (r = 0; r < width; r++) {
    first = last;
    last = new;
    for (j = (0x01 << r); j < max_mask; j <<= 1) {
        for (i = first; i < last; i++) {
            if (buffer[i] >= j) {
                break;
            }
            buffer[new++] = j | buffer[i];
        }
    }
}
```

The goal of the “candidate selection” stage is to obtain a sequence of the operands
that will be XORed to implement the NLF. It starts with an empty sequence and, as a
candidate is selected, it is added to the resulting sequence. The algorithm traverses the
candidate list obtaining the result of the LUT for the candidate. This result is compared
to the result of the sequence generated till that moment when the input is the one defined
by the candidate. If the sequence and the candidate results do match, the candidate is
rejected.

We describe the process for the first three candidates \texttt{00000}, \texttt{00001}, \texttt{00010} of KeeLoq,
whose NLF is defined by the LUT \texttt{0x3A5C742E}.

The result for the first candidate \texttt{00000} is the bit 0, which is 0. The sequence, which
is empty, also has result 0, so the candidate is rejected. The second candidate is \texttt{00001},
and the result associated is 1, different from the result of the sequence, which is 0. The
candidate is added to the sequence. The third candidate is \texttt{00010}, whose result is 1. The
result of the current sequence when only the second bit is active is 0, as the only available
operand is not activated. The candidate is added to the sequence.

Listing 6.4: candidate selection implementation

```c
for (i = 0; i < array_size; i++) {
    aux = 0;
    c_element = array[i];
    c_result = ((0x01 << c_element) & word) ? 1 : 0;
    for (j = 0; j < size; j++) {
        if ( (c_element&result[j]) == result[j] ) {
            aux ^= 1;
        }
    }
    if (c_result != aux) {
```
result[size++] = c_element;
} 
}

The resulting sequence for KeeLoq is 00001, 00010, 00110, 01001, 01100, 10001, 10100, 10011, 10101, 11010, 11100 which matches exactly the ANF presented in Section 6.1. The bitslice implementation can be obtained following the restrictions described in the proposal about the availability of the operands involved in the operations. The methodology is implemented as a compiler optimization with the following steps:

- Control flow graph: obtain the different paths the algorithm can follow in order to traverse them during the following analysis.
- Variable liveness between iterations: in order to detect the input data of the algorithm, the variables that are rotated, which contain the input data of the algorithm that can be manipulated in advance.
- Masking analysis: it keeps a status of the data bits available in the variable, which are not set or cleared by a constant in the algorithm. The NLFSR path should be masked finally to have only one valid bit.
- Bit-level data availability: every instruction of the different basic blocks might modify the bit liveness mask, which is calculated when Boolean operations apply. If another operation is performed, and more than one bit of the variable is being used, this instruction and the instructions that depend on it are not taken into account for the optimization anymore and they will remain in an inner loop with operations that calculate data for one round.
- Sorting: commutative instructions with more bits available in their operands are calculated first and the data availability is recalculated.
- Instruction extraction: instructions with operands with all the data bits available can be extracted from the implementation. The shift is not done on the data value but on the output variable.

6.5 Conclusions

The KeeLoq algorithm is used extensively in RKE applications although it is vulnerable to side-channel analysis. Software implementations based on LUTs have been broken using SPA and CPA. We propose a new software implementation with resistance against timing analysis and CPA based on bitslicing. Bitslice is a non-conventional way to implement algorithms using a scalar processor as a SIMD. We break down the algorithm into logical bit operations so that $N$ parallel operations are possible on a single $N$-bit microprocessor using the ANF representation of KeeLoq. We have first proposed a version not vulnerable to timing analysis with an improvement factor of at least 2.24 when compared to LUT implementations. We have secondly presented a secured version that
also shows resistance against CPA with a mean overhead factor of 2.47 in execution cycles compared to the bitslice implementation. Compared to the LUT-based implementations used in PIC microcontrollers, our proposal provides an improvement factor of 1.21.

We know from [65] that LUT implementations are vulnerable, and we confirm it with our experiments, with less than 200 samples. The bitslice implementation proposed is vulnerable to CPA while the sec-bitslice implementation proposed is proved to be significantly more secure, as 50000 traces are not enough to break the algorithm.
Chapter 7

Conclusions

7.1 Conclusions

FSR-based algorithms are included in security components of portable wireless digital communication and remote keyless entry systems that must cipher every bit transmitted without a significant performance impact, battery-life reduction or buffering needs. The execution of an algorithm in a real device, if not protected, might leak critical information through a side-channel, such as the power consumption of the device while manipulating a secret data. SCA is a family of attacks that exploits the information leaked. Embedded devices, that are accessible to an attacker, are target of these attacks.

Countermeasures against side-channel attacks range among a large variety of solutions. Protecting implementations against physical attacks intends to make the attacks harder. The implementation cost of a countermeasure is of primary importance and must be evaluated with respect to the additional security obtained. The cost, which must be kept low for WSN, includes the effort of engineering as well as the hardware fabrication. There are many hardware countermeasures published, which have the drawback of the resources needed, its cost and the impossibility to adapt to already deployed systems.

There have been successful practical attacks performed over FSR-based algorithms. However, there are only few countermeasure proposals for these algorithms, and most of them are hardware countermeasures. The objective of this Ph.D. thesis is to provide countermeasures to increase the resistance against side-channel attacks of FSR-based ciphers on embedded systems.

In this thesis we propose and evaluate low effort SCA hiding countermeasures to increase the security of FSR-based implementations in embedded systems, suitable to be applied to existing architectures.

We tackle the problem by introducing randomness in the execution of the algorithms using resources available. We have presented a framework to evaluate the security of implementations with customizable blocks: compiler, possible optimizations, target and attack. We have selected two possible targets: a basic MSP430 processor and a generic CPU with a SORU2 co-processor. We have developed cycle accurate simulators for both of them, in Java and SystemC respectively. The leakage model of the targets have been
selected to provide maximum information to the attacker, matching the leakage model of the corresponding attack. In the case of SORU2 we use the Hamming Distance model of the registers of the datapath, traditionally used in hardware implementations. For the MSP430, we use the Hamming Weight model of the registers of the MSP430, typically used in software implementations.

The FSR-based algorithm selected is KeeLoq, widely deployed in RKE applications in both hardware and software implementations. In Section 3.1 we have described the published SCA applied to real KeeLoq implementations. We have used our framework to successfully attack KeeLoq software implementations, using CPA, the attack used in the published attacks. However, we have detected that “ghost peaks” appear, mainly because in FSR-based algorithms, including KeeLoq, the state varies from one round to the next only in one bit. The algorithm manipulates data very similar to the target intermediate value at many different instants. The “ghost peaks” can be detected by visually detecting the window of interest for the attack and ignore other instants of the power trace. However, for the purpose of an automatic evaluation of several software implementations, narrowing the window to the target instant is not trivial. We have proposed an improvement for CPA, which we have named Differential CPA (DCPA), that reduces the effect of “ghost peaks”. In Section 3.4 we have presented this attack and we have shown its efficiency.

We have proposed three countermeasures: first, configuring SORU2, a reconfigurable vector co-processor; second, using standard compiler optimizations; third, using a bitslice implementation of the algorithm, which can be obtained automatically using an optimization pass.

For the first countermeasure we have contributed to the design and development of SORU2. SORU2 is a pipelined reconfigurable vector co-processor. It consists of 4 reconfigurable units (BRUs) connected in cascade through dedicated registers, and vector load and store units that manage the transfer of data between main memory and SORU internal register file. We have developed SORU in order to reduce energy consumption on wireless nodes executing algorithms, in order to implement distributed systems in WSN. SORU2 provides a speedup ranging from 2 to 4.5, reducing the energy consumption up to 80%, when executing vector and matrix benchmarks. We have proposed a KeeLoq configuration for nodes with SORU2. The basic KeeLoq configuration for SORU2 lasts 530 cycles and it is very vulnerable to CPA, as the correlation coefficient for the correct key guess is always 1. We have proposed configurations with almost no timing overhead that reduces the vulnerability of the implementation to CPA. We have proposed to execute several KeeLoq algorithms simultaneously, being only one the correct one and discarding the other results. Our experiments show that this approach, using random input data for the dummy executions, is less efficient than using the same input data with a different key. The random combination of the proposed configurations further improves the security against SCA.

For the second countermeasure, we have evaluated the effect of standard compiler optimizations on an unprotected KeeLoq software implementation. Our hypothesis is that the compiler optimization passes cannot be evaluated individually, outside the context of an optimization sequence. Therefore, we have developed an algorithm to
select the optimization sequences to be analyzed from the unaffordable amount of possible optimization sequences. We conclude from the analysis that there is no generic optimization sequence that makes and insecure implementation to be secure. There are optimization sequences that provide more robust implementations than others, so an analysis to select the best optimization sequence is required. We have confirmed that an analysis on individual optimization pass do not provide enough information, as the `instcombine` optimization pass appears 5 times in the original optimization sequence and its effects on the performance and the DCPA vary from one to the other.

We have proposed the combination of implementations generated by applying different compiler optimization sequences as a countermeasure. The analysis done on the effect of the application of optimization sequences is required to appropriately select the implementations to be combined.

The combination of variable rounds to unroll has been proven to be a good combination, although it is not a completely secure implementation.

The third countermeasure proposed is a new software implementation with resistance against timing analysis and CPA based on bitslicing. Bitslice is a non-conventional way to implement algorithms using a scalar processor as a SIMD. We break down the algorithm into logical bit operations so that \( N \) parallel operations are possible on a single \( N \)-bit microprocessor using the ANF representation of KeeLoq. We have first proposed a version not vulnerable to timing analysis with an improvement factor of at least 2.24 when compared to LUT implementations. We have secondly presented a secured version that also shows resistance against CPA with a mean overhead factor of 2.47 in execution cycles compared to the `bitslice` implementation. Compared to the LUT-based implementations used in PIC microcontrollers, our proposal provides an improvement factor of 1.21.

We have confirmed with our experiments that, following the directions indicated in [65] the KeeLoq LUT software implementation is vulnerable. We have defeated the LUT implementation with less than 200 samples with our framework. The `bitslice` implementation proposed is even more vulnerable to CPA. We have proposed `sec-bitslice`, which randomly selects the amount of rounds executed in parallel. The implementation always executes the maximum amount of rounds (16 for MSP430), although the results of the rounds exceeding the selected number are disposable. In order to introduce noise, the key bits that belong to the disposable rounds are masked with a random mask, executing KeeLoq with a false key. The implementation randomizes the instant when the intermediate value is manipulated and it introduces noise that comes from the execution of KeeLoq with an incorrect key. The algorithm that selects the number of disposable bits is done avoiding any difference in the execution path between the different options. Therefore, SCARE techniques are difficult to apply to this implementation, as the sequence of executed instructions is always kept the same. The `sec-bitslice` is proved to be significantly more secure, as 50000 traces are not enough to break the algorithm.

We have presented three different alternatives to introduce randomness in the execution of KeeLoq, a FSR-based algorithm, using available resources in embedded systems.
Chapter 7. Conclusions

7.2 Future Work

The most important extension of the work presented in this Ph.D thesis would be to complete the evaluation framework. In order to have a completely automatic process, the framework needs a feedback aggregation function to make a decision on the best combination of source code and compiler optimization sequence. We have evaluated performance, code size, DSCA resistance and timing leakage, and obtained a result that should be combined to make the decision.

The framework includes support for executing the evaluated implementations in a simulator and in a real device. The simulator used can be configured to be a high-level target independent simulator, using LLVM interpreter, or a more precise target simulator with cycle accuracy. The model used for the latter option does not aggregate the contributions of the different elements of the target device: buses, functional units, registers, peripherals, etc. In order to consider a worst case scenario, we have considered only the leakage of registers, where data is manipulated. A methodology to extract a suitable model from a given target device would be interesting to provide more flexibility to the framework.

Regarding the SORU2 co-processor, we have provided compiler support to load and control the SORU configurations. However, the configurations loaded have not been automatically generated from a high-level definition of the algorithm. The configurations have been developed in VHDL for Spartan-6 FPGA, considering that the internal logic of the SORU2 BRUs could be implemented with this technology. As the SORU2 co-processor has not been fabricated yet, the BRU has not been defined with precision. If it is finally fabricated, it would be necessary to concrete its implementation.

Finally, we have implemented an optimization pass to automatically generate a bit-slice implementation of FSR-based algorithms. Currently, the bitslice implementation is generated if the optimization pass is loaded. One possibility to improve the system is to provide compiler support to decide the suitability of generating the bitslice implementation considering performance.
Appendices
Appendix A

LLVM optimization passes

In Chapter 5 we evaluate the effect of sequences of compiler optimization passes which are obtained as subsequences of an original sequence of compiler optimization passes. The names of the subsequences refer to the original sequence and they follow the pattern o_XXX_[pn][ai].

The following examples use an hypothetical original optimization sequence with 5 optimization passes to clarify this notation:

\[
\begin{align*}
o_{003}pa & : -opt1 -opt2 -opt3 \\
o_{003}pi & : -opt3 \\
o_{003}ni & : -opt1 -opt2 -opt4 -opt5 \\
o_{003}na & : -opt4 -opt5
\end{align*}
\]

The original optimization sequence used in Chapter 5, from which the algorithm extracts the subsequences to be analyzed, is obtained from the following command:

```
llvm-as < /dev/null | \ 
  opt -std-compile-opts -disable-output -debug-pass=Arguments
```

The sequence consists of 65 optimization passes. We enumerate them to provide information about the optimization passes included in the subsequences evaluated:

\[
\begin{align*}
1 & : -targetlibinfo \\
2 & : -no-aa \\
3 & : -tbaa \\
4 & : -basicaa \\
5 & : -preverify \\
6 & : -domtree \\
7 & : -verify \\
8 & : -globalopt \\
9 & : -ipsccp \\
10 & : -deadargelim \\
11 & : -instcombine \\
12 & : -simplifycfg \\
13 & : -basiccg \\
14 & : -prune-eh \\
15 & : -inline \\
16 & : -functionattrs \\
17 & : -argpromotion \\
18 & : -scalarrepl-ssa \\
19 & : -domtree \\
20 & : -early-cse \\
21 & : -simplify-libcalls \\
22 & : -lazy-value-info \\
23 & : -jump-threading \\
24 & : -correlated-propagation \\
25 & : -simplifycfg \\
26 & : -instcombine \\
27 & : -tailcalleelim \\
28 & : -simplifycfg
\end{align*}
\]
Appendix A. LLVM optimization passes

29 -reassociate
30 -domtree
31 -loops
32 -loop-simplify
33 -lcssa
34 -loop-rotate
35 -licm
36 -lcssa
37 -loop-unswitch
38 -instcombine
39 -scalar-evolution
40 -loop-simplify
41 -lcssa
42 -indvars
43 -loop-idiom
44 -loop-deletion
45 -loop-unroll
46 -memdep
47 -gvn
48 -memdep
49 -memcpyopt
50 -sccp
51 -instcombine
52 -lazy-value-info
53 -jump-threading
54 -correlated-propagation
55 -domtree
56 -memdep
57 -dse
58 -adce
59 -simplifcfg
60 -instcombine
61 -strip-dead-prototypes
62 -globaldce
63 -constmerge
64 -preverify
65 -domtree
Table A.1 shows the performance evaluation of the 51 implementations generated by the selection algorithm after CFG filtering in Chapter 5.

<table>
<thead>
<tr>
<th></th>
<th>exec cycles</th>
<th>Code size (bytes)</th>
<th>Register usage</th>
<th>Memory usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>max</td>
<td></td>
<td>static</td>
</tr>
<tr>
<td>o_018_pa</td>
<td>33233</td>
<td>34147</td>
<td>200</td>
<td>10</td>
</tr>
<tr>
<td>o_020_pa</td>
<td>33233</td>
<td>34147</td>
<td>200</td>
<td>10</td>
</tr>
<tr>
<td>o_025_pa</td>
<td>33316</td>
<td>34230</td>
<td>204</td>
<td>10</td>
</tr>
<tr>
<td>o_018_pi</td>
<td>33761</td>
<td>34675</td>
<td>204</td>
<td>10</td>
</tr>
<tr>
<td>o_011_ni</td>
<td>36066</td>
<td>36931</td>
<td>220</td>
<td>11</td>
</tr>
<tr>
<td>o_012_na</td>
<td>36066</td>
<td>36931</td>
<td>220</td>
<td>11</td>
</tr>
<tr>
<td>o_015_na</td>
<td>36066</td>
<td>36931</td>
<td>220</td>
<td>11</td>
</tr>
<tr>
<td>o_012_ni</td>
<td>42850</td>
<td>43849</td>
<td>244</td>
<td>11</td>
</tr>
<tr>
<td>o_015_ni</td>
<td>42850</td>
<td>43849</td>
<td>244</td>
<td>11</td>
</tr>
<tr>
<td>o_020_ni</td>
<td>42850</td>
<td>43849</td>
<td>244</td>
<td>11</td>
</tr>
<tr>
<td>o_026_ni</td>
<td>42850</td>
<td>43849</td>
<td>244</td>
<td>11</td>
</tr>
<tr>
<td>o_027_ni</td>
<td>42850</td>
<td>43849</td>
<td>244</td>
<td>11</td>
</tr>
<tr>
<td>o_029_ni</td>
<td>42850</td>
<td>43849</td>
<td>244</td>
<td>11</td>
</tr>
<tr>
<td>o_034_pa</td>
<td>42850</td>
<td>43849</td>
<td>244</td>
<td>11</td>
</tr>
<tr>
<td>o_035_ni</td>
<td>42850</td>
<td>43849</td>
<td>244</td>
<td>11</td>
</tr>
<tr>
<td>o_035_pa</td>
<td>42850</td>
<td>43849</td>
<td>244</td>
<td>11</td>
</tr>
<tr>
<td>o_038_pa</td>
<td>42850</td>
<td>43849</td>
<td>244</td>
<td>11</td>
</tr>
<tr>
<td>o_041_pa</td>
<td>42850</td>
<td>43849</td>
<td>244</td>
<td>11</td>
</tr>
<tr>
<td>o_042_pa</td>
<td>42850</td>
<td>43849</td>
<td>244</td>
<td>11</td>
</tr>
<tr>
<td>o_047_pa</td>
<td>42850</td>
<td>43849</td>
<td>244</td>
<td>11</td>
</tr>
<tr>
<td>o_026_pa</td>
<td>43499</td>
<td>44632</td>
<td>252</td>
<td>11</td>
</tr>
<tr>
<td>o_027_pa</td>
<td>43499</td>
<td>44632</td>
<td>252</td>
<td>11</td>
</tr>
<tr>
<td>o_029_pa</td>
<td>43499</td>
<td>44632</td>
<td>252</td>
<td>11</td>
</tr>
<tr>
<td>o_033_pa</td>
<td>43499</td>
<td>44632</td>
<td>252</td>
<td>11</td>
</tr>
<tr>
<td>o_034_ni</td>
<td>43549</td>
<td>44682</td>
<td>252</td>
<td>11</td>
</tr>
<tr>
<td>o_018_ni</td>
<td>49287</td>
<td>50621</td>
<td>276</td>
<td>10</td>
</tr>
<tr>
<td>o_047_pi</td>
<td>61458</td>
<td>63690</td>
<td>312</td>
<td>7</td>
</tr>
<tr>
<td>o_009_pa</td>
<td>62529</td>
<td>65610</td>
<td>300</td>
<td>7</td>
</tr>
<tr>
<td>o_012_pi</td>
<td>62529</td>
<td>65610</td>
<td>300</td>
<td>7</td>
</tr>
<tr>
<td>o_015_pi</td>
<td>62529</td>
<td>65610</td>
<td>300</td>
<td>7</td>
</tr>
<tr>
<td>o_020_pi</td>
<td>62529</td>
<td>65610</td>
<td>300</td>
<td>7</td>
</tr>
<tr>
<td>o_023_pi</td>
<td>62529</td>
<td>65610</td>
<td>300</td>
<td>7</td>
</tr>
<tr>
<td>o_029_pi</td>
<td>62529</td>
<td>65610</td>
<td>300</td>
<td>7</td>
</tr>
<tr>
<td>o_035_pi</td>
<td>62529</td>
<td>65610</td>
<td>300</td>
<td>7</td>
</tr>
<tr>
<td>o_050_pi</td>
<td>62529</td>
<td>65610</td>
<td>300</td>
<td>7</td>
</tr>
<tr>
<td>o_000</td>
<td>62529</td>
<td>65610</td>
<td>300</td>
<td>7</td>
</tr>
</tbody>
</table>

Continued on next page
### Table A.1 – continued from previous page

<table>
<thead>
<tr>
<th>Procedure</th>
<th>exec cycles</th>
<th>Code size (bytes)</th>
<th>Register usage</th>
<th>Memory usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>o_034_pi</td>
<td>62543-65490</td>
<td>316</td>
<td>7</td>
<td>0-70</td>
</tr>
<tr>
<td>o_011_pa</td>
<td>63067-65880</td>
<td>312</td>
<td>7</td>
<td>0-70</td>
</tr>
<tr>
<td>o_011_pi</td>
<td>63067-65880</td>
<td>312</td>
<td>7</td>
<td>0-70</td>
</tr>
<tr>
<td>o_012_pa</td>
<td>63067-65880</td>
<td>312</td>
<td>7</td>
<td>0-70</td>
</tr>
<tr>
<td>o_015_pa</td>
<td>63067-65880</td>
<td>312</td>
<td>7</td>
<td>0-70</td>
</tr>
<tr>
<td>o_051_na</td>
<td>63067-65880</td>
<td>312</td>
<td>7</td>
<td>0-70</td>
</tr>
<tr>
<td>o_053_na</td>
<td>63067-65880</td>
<td>312</td>
<td>7</td>
<td>0-70</td>
</tr>
<tr>
<td>o_018_na</td>
<td>67091-69055</td>
<td>324</td>
<td>7</td>
<td>0-70</td>
</tr>
<tr>
<td>o_023_na</td>
<td>67091-69055</td>
<td>324</td>
<td>7</td>
<td>0-70</td>
</tr>
<tr>
<td>o_026_na</td>
<td>67091-69055</td>
<td>324</td>
<td>7</td>
<td>0-70</td>
</tr>
<tr>
<td>o_028_na</td>
<td>67091-69055</td>
<td>324</td>
<td>7</td>
<td>0-70</td>
</tr>
<tr>
<td>o_029_na</td>
<td>67091-69055</td>
<td>324</td>
<td>7</td>
<td>0-70</td>
</tr>
<tr>
<td>o_038_na</td>
<td>67091-69055</td>
<td>324</td>
<td>7</td>
<td>0-70</td>
</tr>
<tr>
<td>o_034_na</td>
<td>67685-69783</td>
<td>328</td>
<td>7</td>
<td>0-70</td>
</tr>
<tr>
<td>o_035_na</td>
<td>67691-69789</td>
<td>328</td>
<td>7</td>
<td>0-70</td>
</tr>
<tr>
<td>o_047_na</td>
<td>70379-73326</td>
<td>340</td>
<td>7</td>
<td>0-70</td>
</tr>
</tbody>
</table>

*Table A.1: Performance evaluation*
This annex provides the results of CPA applied to the implementation groups generated in Chapter 5. There are four groups of figures: CPA and resistance calculation on the whole traces, and CPA and resistance calculation on the time window when the target intermediate value can be manipulated, represented by blue dashed lines in the global CPA figures.
Figure A.1: CPA on keeloq_tb041 generated implementations
Figure A.1: CPA on keeloq_tb041 generated implementations