Lecture Notes:
Hardware and Embedded Systems Security

David Oswald

April 27, 2023

Table of Contents

General Remarks

Notation

Notation and symbols are (should be) consistent within one chapter. However, between chapters, the notation may change (e.g., n could both refer to the block size of a block cipher and the RSA modulus. Some symbols should be consistent throughout the document, for example p for plaintext, c for ciphertext, and k for the key, unless explicitly re-defined for a chapter or section.

Byte and bit indices are counted from zero, i.e., the Least Significant Bit (LSB) in the Least Significant Byte (LSByte) is byte 0, bit 0. In binary notation (e.g., $19_{10} = (10011)_2$), the LSB is at the right.

Bug Reports

If you find a problem, bug, or error, please let me know (include the page number and a brief description): d.f.oswald@cs.bham.ac.uk—thanks! You can also submit a pull request on the lecture notes repo at `https://github.com/david-oswald/hwsec_lecture_notes`.

Licensing

These lectures notes are under the Creative Commons "Attribution-ShareAlike 3.0 Unported" (`CC BY-SA 3.0`) license. See `https://creativecommons.org/licenses/by-sa/3.0/` for details. This means that ou are free to:

Share Copy and redistribute the material in any medium or format.

Adapt Remix, transform, and build upon the material for any purpose, even commercially.

Under the following terms:

Attribution You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

Chapter 1

Introduction

1.1 Hardware and Embedded Systems Security

Embedded devices have become commonplace in our world—billions of small (or today also more powerful) processors are invisibly integrated in objects of our daily life. Examples include vehicles, contactless payment and identity cards, smartphones, electronic locking systems, industrial machines, trains, traffic control, medical devices, and a countless number of other systems. The infamous term Internet of Things (IoT) has gained significant attention in the past years, implying that these pervasive computing devices communicate and form a global network.

With this trend grows the need for security of the respective devices—threats are manifold, and in contrast to many PC-based applications, security incidents can easily have direct influence on the physical world (consider for example a vulnerability in a self-driving car). Evaluating and improving the security of such devices is the core problem of the field of "Hardware and Embedded Systems Security".

In this lecture, we look at both constructive (i.e. "how to create secure embedded systems?") and destructive aspects ("how to attack embedded systems?"). The first part deals with the efficient implementation of different cryptographic algorithms on constrained devices (Chapter 2 and Chapter 3). The second part covers "implementation attacks", a class of attacks that allows to break mathematically secure algorithms by exploiting physical characteristics of a real device.

1.2 Development Platform Used for this Lecture

For the purposes of this lecture, we will use the MSP-EXP430FR5969 launchpad[1] as both implementation and Side-Channel Analysis (SCA) evaluation platform. The board features a MSP430FR5969 16 MHz ultra-low-power Microcontroller (µC) with 64 kB of Ferroelectric RAM (FRAM) for storage of both volatile data and the program code [Tex16]. Acknowledgments go to Texas Instruments for providing the boards as part of their university program.

The MSP430 architecture is part of the lower end of embedded devices, with a 16-bit Reduced Instruction Set Computer (RISC) architecture and limited computational capabilities. However, such processors are very cost effective and feature a very low power consumption, which is why such µCs (and other similar processors, e.g., Microchip PIC or Atmel ATMega and ATXmega µCs) are used in billions of embedded systems across numerous application areas.

Using such a processor for educational purposes has several advantages: first, due to the (relative) simplicity, the developer has full control over the system, e.g., can optimize an algo-

[1] http://www.ti.com/product/MSP430FR5969

rithm on the assembly level and estimate the precise cycle count. In addition, techniques that incur large memory overhead are automatically excluded—programs have to be developed with speed requirements and size considerations in mind. Finally, applying implementation attacks is easier compared to large processors, where an Operating System (OS) schedules several tasks or where several processes execute in parallel.

Chapter 2

Implementation of Symmetric (Secret Key) Cryptography

The first part of this lecture deals with the (efficient) implementation of symmetric cryptographic algorithms, in particular *block ciphers*. However, most of the concepts also apply to other primitives (hash functions, stream ciphers, Message Authentication Codes (MACs)) in a similar manner.

2.1 Block Ciphers

A block cipher is a function $bc\,(p,\,k)$ that maps ("encrypts") a plaintext p to a ciphertext c, given a key k. For given k, bc is invertible (i.e., we can decrypt: $p = bc^{-1}\,(c,\,k)$). Both plaintext p and ciphertext c are of fixed *block size* n (measured in bit). The key is of fixed *key size* m. A high-level block diagram for a typical block cipher is shown in Fig. 2.1.

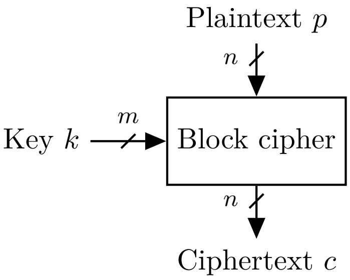


Figure 2.1: High-level view of a block cipher

A few examples for typical block ciphers with their respective key and block sizes are:

AES-128 The Rijndael [DR99] cipher with $n = m = 128$.

DES The Data Encryption Standard [NIS] with $n = 64$, $m = 56$.

3DES Three DES in Encrypt-Decrypt-Encrypt (EDE) configuration, $n = 64$, $m = 112$ or $m = 168$ (for 2-key and 3-key 3DES).

PRESENT A lightweight block cipher [BKL+07] with $n = 64$ and $m = 80$ or $m = 128$ (depending on selected key length).

Block ciphers are usually *iterated* ciphers, i.e., a *round function* is applied multiple times to map plaintext to ciphertext. Each round function operates on the *state* of the cipher, that is usually initialized with the plaintext p and hence of block size n. In a particular round i, a *round key* k_i (derived from the key k using the *key schedule*) enters the round function together with the current state s_i. The final state is then the ciphertext. Commonly, the round function of a block cipher follow one of two designs (although there are of course other approaches not covered here): the *Feistel* design principle (DES, 3DES, cf. Fig. 2.2) or the (newer) Substitution-Permutation Network (SPN) approach (AES, PRESENT, cf. Fig. 2.3).

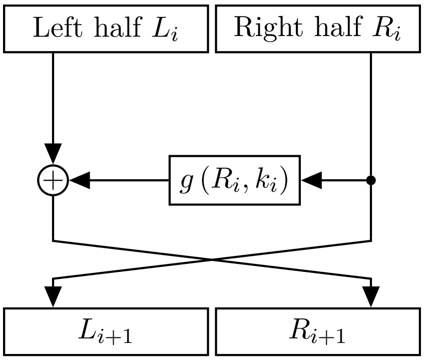


Figure 2.2: One round of a (balanced) Feistel cipher

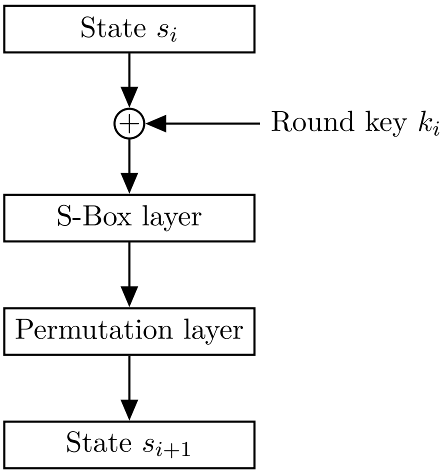


Figure 2.3: One round of an SPN cipher (note that the key addition may also happen at the end of the round)

2.1.1 Building Blocks

To provide cryptographic security, block ciphers require elements to provide "diffusion" ("a single input bit change affects all output bits with 50% probability") and "confusion" ("the relationship between input and output is sufficiently complex"). The core elements to reach these goals in virtually all block ciphers are:

Key addition The key is combined with the plaintext / the state through an addition-like operation, normally bitwise Exclusive OR (XOR) or arithmetic addition over a finite ring (e.g., mod 2^{32}).

S-Box A Substitution Box (S-Box) is a non-linear mapping, usually over a small number of bits (e.g. 8-to-8 or 6-to-4). A cipher may have multiple, different S-Boxes (like the DES) or one one single S-Box (e.g., AES or PRESENT). The S-Box may be (seemingly) chosen at random (DES, PRESENT) or follow an algebraic structure (AES). Provides "confusion".

Permutations A linear mapping to provide "diffusion", permuting the output of the S-Box layer such that one S-Box output bit affects multiple S-Box inputs in the next round. Can be realized bitwise (like in DES, PRESENT) or through bytewise/wordwise operations (like in the AES).

To give an example, the PRESENT block cipher ($n = 64$, $m = 80$) is given in the following pseudocode:

Algorithm 1 PRESENT (pseudocode based on [BKL+07])

```
$s \leftarrow p$
$k_i \leftarrow$ generateRoundKeys($k$)
for i = 1 ... 31 do
    $s \leftarrow$ addRoundKey($s$, $k_i$)
    $s \leftarrow$ sboxLayer($s$)
    $s \leftarrow$ pLayer($s$)
end for
$c \leftarrow$ addRoundKey($s$, $k_{32}$)
```

The `addRoundKey` operation is simple bitwise XOR of the state s and the round key k_i. The `sboxLayer` operation applies the 4-to-4 PRESENT S-Box to the state in groups of 4 bit each, i.e., the S-Box is applied a total of $^{64}/_{4} = 16$ times. The PRESENT S-Box is given in the following Table 2.1 (as hexadecimal digits):

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Table 2.1: PRESENT S-Box

The `pLayer` operation is a bitwise permutation, whereas the bit at position j is permuted to position

$$j_p = \left\lfloor \frac{j}{4} \right\rfloor + (j \bmod 4) \times 16$$

i.e., bit 0 is permuted to position 0, bit 1 to 16, bit 2 to 32, bit 3 to 48, bit 4 to 1, and so on. The original paper [BKL+07] supplies this as a full table. We will discuss the cost of each of these core operations (in software and hardware) in the following, using the example of PRESENT again.

A Note on Bitwise Operations in `C` To implement bitwise permutations (and other bitwise manipulations), usually two primitive operations are required: getting the value of a bit in a word (`getbit`) and setting/clearing a bit in a word (`setbit` and `clrbit`). In `C`, assuming the variable `word` is a byte (type `uint8_t`), this can be implemented as follows:

```
uint8_t getbit(const uint8_t word, const uint8_t bit)
{
    return (word >> bit) & 0x1;
    // or, if return value != 0 means: bit set
    // return (word & (1 << bit));
}

void setbit(const uint8_t* word, const uint8_t bit)
{
    *word |= (1 << bit);
}

void clrbit(const uint8_t* word, const uint8_t bit)
{
    *word &= ~(1 << bit);
}
```

Note that this is the most straightforward approach and can be often optimized. Also, note that using dedicated functions for this task can cause significant runtime overhead, therefore, usually macros are used for these operations. On some platforms, there are dedicated assembly instructions for manipulating bits in a register, e.g., `SBR` and `CBR` to set and clear bits in a register in the Atmel AVR architecture[1], as well as the equivalent for the MSP430 with `BIS` and `BIC`[2] or the ARM Cortex M0 with `ANDS`, `ORS`, `BICS`[3].

These specific instructions usually avoid the load, update, store cycle required when directly translating the above `C` code. Note that optimizing compilers for these architectures often recognize the above constructs and automatically replace them with the appropriate assembly instruction.

[1] http://www.atmel.com/webdoc/avrassembler/avrassembler.wb_SBR.html and http://www.atmel.com/webdoc/avrassembler/avrassembler.wb_CBR.html

[2] https://www.ti.com/sc/docs/products/micro/msp430/userguid/as_5.pdf

[3] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0497a/CIHJJEIH.html

2.1.2 Key Addition Layer

Since the key addition is, as mentioned, usually XOR or rarely arithmetic addition, it is cheap both in hardware (XOR gates are a basic building block) and software (nearly every processor has dedicated `XOR` and `ADD` instructions). Due to the simplicity of the operation, there is usually no or very little optimization potential (i.e., by unrolling the loop that performs the XOR) at all.

2.1.3 S-Box Layer

In software implementations, the S-Box is normally realized as a Look-Up Table (LUT) (i.e., an array stored in Random Access Memory (RAM) or Flash) and hence normally fast. However, the following optimizations (and probably more) may be considered:

- Ciphers with multiple, different S-Boxes (e.g., DES) have higher memory requirements and are hence less suited for compact software implementations.
- To optimize for execution time, it may be beneficial to merge multiple S-Box instances into one look-up, e.g., for PRESENT, two 4-to-4 lookups can be done in parallel by using an 8-to-8 LUT.
- Depending on the hardware platform, certain memories can be faster. For example, a read from RAM may take one cycle, while a Flash access needs two cycles. Hence, it could be beneficial to load the S-Box into RAM (if sufficient space is available) before carrying out encryption.
- If memory complexity is to be minimized, a smaller S-Box is generally better. For example, to store a a-to-b bit S-Box, $b \times 2^a$ bit storage are needed at least—for example, for PRESENT, the 4-to-4 S-Box needs at least 64 bit = 8 byte to be stored. Note that, however, most architectures do not allow addressing below the byte level, so additional (time) overhead will be incurred to correctly apply the S-Box. Commonly, S-Boxes are not larger than 8-to-8 (as in AES), which already requires 256 byte for storage.

To further illustrate the last point, consider the 4-to-4 PRESENT S-Box. In `C` syntax, this S-Box would be stored in an array, and a look-up performed on a single state byte `s` as in the following example:

```
const uint8_t sbox[16] = { 0xC, 0x5, 0x6, 0xB, 0x9, 0x0, 0xA, 0xD,
    0x3, 0xE, 0xF, 0x8, 0x4, 0x7, 0x1, 0x2 };

// Look-up lower nibble
uint8_t s_new = sbox[s & 0x0F];

// Look-up upper nibble
s_new |= sbox[(s >> 4) & 0x0F] << 4;

// Update state
s = s_new;
```

However, the additional bit shifts and boolean operations create significant overhead. Hence, the idea is to merge two instances of the S-Box by using a larger, 256-byte table of the following structure ($sb\left(in\right)$ is the 4-bit output value, $|$ indicates concatenation of two nibbles to a full byte):

$$\begin{pmatrix} sb\left(0x0\right)\mid sb\left(0x0\right) & sb\left(0x0\right)\mid sb\left(0x1\right) & sb\left(0x0\right)\mid sb\left(0x2\right) & \dots & sb\left(0x0\right)\mid sb\left(0xF\right) \\ sb\left(0x1\right)\mid sb\left(0x0\right) & sb\left(0x1\right)\mid sb\left(0x1\right) & sb\left(0x1\right)\mid sb\left(0x2\right) & \dots & sb\left(0x1\right)\mid sb\left(0xF\right) \\ sb\left(0x2\right)\mid sb\left(0x0\right) & sb\left(0x2\right)\mid sb\left(0x1\right) & sb\left(0x2\right)\mid sb\left(0x2\right) & \dots & sb\left(0x2\right)\mid sb\left(0xF\right) \\ & & \dots & & \\ sb\left(0xF\right)\mid sb\left(0x0\right) & sb\left(0xF\right)\mid sb\left(0x1\right) & sb\left(0xF\right)\mid sb\left(0x2\right) & \dots & sb\left(0xF\right)\mid sb\left(0xF\right) \end{pmatrix}$$

In (abbreviated) `C`, this translates to:

```
const uint8_t sbox[256] = {
        0xCC, 0xC5, 0xC6, ..., 0xC1, 0xC2, // up: sb(0x0) = 0xC
        0x5C, 0x55, 0x56, ..., 0x51, 0x52, // up: sb(0x1) = 0x5
        ...
        0x1C, 0x15, 0x16, ..., 0x11, 0x12, // up: sb(0xE) = 0x1
        0x2C, 0x25, 0x26, ..., 0x21, 0x22, // up: sb(0xF) = 0x2
};

// Look-up two nibbles in parallel
s = sbox[s]
```

In hardware, there are two basic ways to realize an S-Box: as a LUT (using RAM or other memories) or in combinatorial logic. In the latter case, each output bit is represented as a Boolean expression in the input bits. These functions are then implemented using logic gates (e.g., `AND`, `NAND`, `OR`, `NOR`, `XOR`). While we do not focus on hardware implementations in this lecture, this concept is explained here since it has relevance for certain software optimizations as well.

Taking a 4-to-4 S-Box as an example, this results in four Boolean expressions:

$$\begin{aligned} r_0 &= sb_0\left(in_0, in_1, in_2, in_3\right) \\ r_1 &= sb_1\left(in_0, in_1, in_2, in_3\right) \\ r_2 &= sb_2\left(in_0, in_1, in_2, in_3\right) \\ r_3 &= sb_3\left(in_0, in_1, in_2, in_3\right) \end{aligned}$$

where $in = in_3 in_2 in_1 in_0$ is the binary representation of the S-Box input and $r = r_3 r_2 r_1 r_0$ the one of the output. The Boolean expressions sb_i can be written using standard techniques, e.g., in Conjunctive Normal Form (CNF) [Wik15b], Disjunctive Normal Form (DNF) [Wik16a], and Algebraic Normal Form (ANF) [Wik15a]. Karnaugh maps [Wik16b] can be used to minimize the circuit complexity.

A simple example for such a Boolean function (unrelated to any real S-Box) in DNF would be:

$$sb_0\left(in_0, in_1, in_2, in_3\right) = \left(in_0 \wedge \overline{in_1} \wedge in_3\right) \vee \left(in_1 \wedge in_2\right) \vee \left(\overline{in_3}\right)$$

Note that it would be also possible to use Boolean expressions in software implementations (computing every output bit separately), however, the overhead both in code size and execution time would be prohibitive in virtually all practical scenarios. Yet, as mentioned, certain software optimizations (see Sect. 2.3) are still based on this approach.

2.1.4 Permutation Layer

The efficiency of permutations in software highly depends on the type of permutation: permutations on the byte (or multiple-of-byte) level are generally fast since the addressing scheme of processors usually allows to copy a single byte. For example, the AES was designed with fast software implementations in mind, hence, the permutation layer (`ShiftRows` and `MixColumns`) operates on a byte or word level. A `C` example for such a byte-wise permutation would be:

```c
// Permute bytes 0, 1, 2, 3 -> 2, 3, 1, 0
const uint8_t tmp = s[0];
s[0] = s[2];
s[2] = s[1];
s[1] = s[3];
s[3] = tmp;
```

In contrast, bit-wise permutations are less efficient to implement, as every bit has to be extracted and copied to its output position. For example, moving the single bit 14 (byte position $\lfloor 14/8 \rfloor = 1$, bit position $(14 \bmod 8) = 6$) to bit 36 (byte 4, bit 4) would be implemented as:

```c
// Byte 1, bit 6 to byte 4, bit 4
// Get source bit
const uint8_t tmp = (s[1] >> 6) & 0x1;
// Assume out is zeroed before
out[4] |= tmp << 4;
```

Note that such operations have to be carried out for each bit (e.g., 64 times in the case of PRESENT or 32 times for DES) and are harder to realize *in-place* (e.g., a separate output buffer is needed).

In contrast, in hardware implementations, permutations are usually "free" since they represent simple interconnects. DES, for example, was designed as a cipher fast in hardware and hence makes extensive use of bitwise permutations. AES was designed for software and implementation on 8-bit (or higher) processors. PRESENT represents a "middle ground" for a very lightweight cipher that is fast or small both in hardware and software (even on extremely constrained 4-bit µCs).

2.2 Table-based Software Optimizations

As pointed out in the previous section, bitwise permutations are in general expensive in software. In this section, we discuss one approach to mitigate this problem. Section 2.3 then discusses an alternative approach that may give further speed-up in certain scenarios. As we have already seen in Section 2.1.3, spending slightly more memory may speed up operations considerably.

The core idea of this section is twofold: On the one hand, we merge the S-Box and permutation layer into one operation. Secondly, we generate tables with output size that is as big as the internal state, i.e., while a normal PRESENT S-Box is 4-to-4 bit, we now use a 4-to-64 bit table. Again, recall that storing and a-to-b bit table requires $b \times 2^a$ bit, i.e., increasing the output size b only linearly affects the overall table size (not exponential like a). Hence, the memory overhead stays within reasonable limits (e.g., $64 \times 2^4 = 1024$ bit instead of $4 \times 2^4 = 64$ bit).

Before looking at a "real" example for the approach of this section, consider an artificial byte-oriented cipher with a 4-byte (32-bit) state. Assume that the same S-Box is applied to each state byte, followed by the byte permutation $(0, 1, 2, 3) \rightarrow (2, 3, 1, 0)$. A straightforward implementation would be:

```
// Apply S-Box
for(uint8_t i = 0; i < 4, i++)
{
    s[i] = sbox[s[i]];
}

// Permute bytes 0, 1, 2, 3 -> 2, 3, 1, 0
const uint8_t tmp = s[0];
s[0] = s[2];
s[2] = s[1];
s[1] = s[3];
s[3] = tmp;
```

However, note that the S-Box and permutation layer can be easily merged in this case, saving four write operations to the state:

```
// Permute bytes 0, 1, 2, 3 -> 2, 3, 1, 0 and apply S-Box
const uint8_t tmp = s[0];
s[0] = sbox[s[2]];
s[2] = sbox[s[1]];
s[1] = sbox[s[3]];
s[3] = sbox[tmp];
```

For ciphers with byte-oriented permutations (like `ShiftRows` in the AES), this optimization has a relatively minor effect, since the permutation layer is in any case computationally efficient. On the other hand, for bitwise permutations, the above method cannot be applied directly since multiple bytes are affected by each S-Box output. This is where "artificially" increasing the table output length to the full state size comes into play: the output bits of the S-Box are directly stored at their *permuted* output positions, while the remaining state bits are set to zero.

To illustrate this concept, consider the PRESENT S-Box for the first state byte s_0. In binary, the S-Box is given as:

$$\begin{aligned}
0xC &= (1100)_2 \\
0x5 &= (0101)_2 \\
0x6 &= (0110)_2 \\
&\dots \\
0x2 &= (0010)_2
\end{aligned}$$

The four output bits after the permutation affected by this S-Box output are bit 0, bit 16, bit 32, and bit 48. In the new table, we store the full width of the state, with the output bits moved to the correct positions:

$$\begin{aligned}
&(0\dots 0\underline{1}0\dots 0\underline{1}0\dots 0\underline{0}0\dots 00\underline{0})_2 \\
&(0\dots 0\underline{0}0\dots 0\underline{1}0\dots 0\underline{0}0\dots 00\underline{1})_2 \\
&(0\dots 0\underline{0}0\dots 0\underline{1}0\dots 0\underline{1}0\dots 00\underline{0})_2 \\
&\dots \\
&(0\dots 0\underline{0}0\dots 0\underline{0}0\dots 0\underline{1}0\dots 00\underline{0})_2
\end{aligned} \tag{2.1}$$

This means that in the output of the table for the first S-Box, only the bits 0, 16, 32, and 48 can ever take a non-zero value. For the second S-Box, there is a *separate*, different table with only the bits 1, 17, 33, and 49 changing, i.e., we have the pattern:

$$(0\dots \underline{x}00\dots \underline{x}00\dots \underline{x}00\dots 0\underline{x}0)_2$$

In total, this yields 16 4-to-64 tables, one for each S-Box. Note that now the table for every S-Box instance is different due to the different permuted positions. The final question is how to re-combine the outputs of each table into one state. This can be easily achieved using bitwise `XOR` or `OR` operations, which both yield the desired result because there is never more than one non-zero bit per bit position[4].

The following `C` code illustrates the implementation of this technique for two S-Boxes:

```c
const uint64_t spbox0[16] = {
    0x0001000100000000ULL, 0x0000000100000001ULL,
    0x0000000100010000ULL, ..., 0x0000000000010000ULL };
const uint64_t spbox1[16] = { ... };
...
// Look-up lower nibble
uint64_t s_new = spbox0[s & 0x0F];
// Look-up upper nibble and XOR into state
s_new ^= spbox1[(s >> 4) & 0x0F];
...
```

[4] As a side question, think about why bitwise `AND` would *not* be suitable.

Why does this work? The key reason is that it is by definition impossible that we have two non-zero bits at the *same position* when combining (`XOR`ing) multiple "S-P-Box" outputs. That is, at every bit position, there is at most one 1 and (in the case of PRESENT) fifteen 0 bits. `XOR` (and also `OR`) has the property that $1 \oplus 0 = 1$ but $0 \oplus 0 = 0$. Obviously, without the guarantee that there is at most one 1 per bit position, the optimization would *not* work.

Efficiency and Cost By combining the S-Box and permutation layer into slightly bigger, individual tables, we gain significant speed-up as bit permutations are completely avoided. The disadvantage lies in the memory requirements. While a straightforward PRESENT S-Boxes consumes $4 \times 2^4 = 64\,\text{bit} = 8\,\text{byte}$, the combined tables require $16 \times 64 \times 2^4 = 16384\,\text{bit} = 2048\,\text{byte}$ of storage (the additional factor 16 is due to the fact that we now have one table per S-Box, rather than one shared LUT). This is a significant overhead, however, of course some amount of program code memory (for implementing the permutation) is saved.

If the size of the table is of concern, note that due to the regular structure of the PRESENT permutation, it would be possible to save memory by storing the combined table once, looking-up a value, and shifting it left by one for each S-Box (i.e. no shift if first S-Box, shift by one for second S-Box, and so on). This single table would then be $64 \times 2^4 = 1024\,\text{bit} = 128\,\text{byte}$ long.

In addition, for PRESENT, further speed improvements could be reached by combining the table-based approach with the "merge two S-Boxes" idea explained in Section 2.1.3. To store these tables, $8 \times 64 \times 2^8 = 131072\,\text{bit} = 16384\,\text{byte} = 16\,\text{kByte}$ would be needed, but the runtime for the combined S-Box look-up and permutation would be approximately halved overall. Again, using the above "shifting" approach, the size could be brought down by a factor of 8 to $64 \times 2^8 = 16384\,\text{bit} = 2048\,\text{byte}$.

For the AES, there is a variant of these table-based optimizations called *T tables*, exploting some properties of the `MixColumns` transform for very fast implementation on 32-bit platforms. This optimization is further explained in the original AES proposal [DR99, Section 5.2].

2.3 Bitslicing

Rather than "working around" the problem of efficiently performing bitwise permutations by using tables, the *bitslicing* method explained in this section takes a radically different approach to the problem of fast implementation of symmetric cryptography in software. First introduced by Biham in [Bih97], it makes use of the fact that often, one has to encrypt many block of plaintext at the same time (e.g., for disk encryption, network transfers, exhaustive key search, to name a few).

The core idea is best explained by revisiting the problem of permutations again—they are fast as long as they are done on the level of the smallest addressable unit on a processor, i.e., normally a byte. In theory, one could realize a bitwise permutation (with significant storage overhead) by storing each bit of the state in a individual byte (leaving the remaining 7 bit set to zero), and then permuting the respective bytes. However, that would mean that one would need—for the example of a cipher with 64-bit state—64 byte, of which only 1/8 are used. Bitslicing allows to "reclaim" the bits that would be "wasted" in the above approach.

However, let us for now follow the approach of storing one bit per byte in order to see how an implementation of the typical components of a block cipher would look like. We assume that both plaintext and the key (round keys) are stored in this form. In `C`, we would the have an structure as in the following example:

```
// Plaintext = 0xF9 ... AC = 11111001 ... 10101100
uint8_t state[64] = {0, 0, 1, 1, 0, 1, 0, 1, ...,
    1, 0, 0, 1, 1, 1, 1, 1 };

uint8_t key[64] = {0, 1, 1, 0, ...};

for(uint8_t round = 1; round <= 31; round++)
{
    // Key addition
    // S–Box
    // Permutation
    // Round key update
}
```

Key Addition Layer Since both round key and state are in the "expanded" form, we can simply XOR them byte-wise, whereas each XOR between two bytes of course only takes care of one bit in the state. The `C` pseudocode would be:

```
for(uint8_t i = 0; i < 64; i++)
{
    state[i] = state[i] ^ roundkey_exp[i];
}
```

Permutation Layer The bitwise permutation of PRESENT now becomes a byte-wise permutation (due to the wasteful way of storing the state bits). For example, for the first few bits of the PRESENT permutation, this would result in (assuming the permutation is not done in-place):

```
state_out[0]  = state[0];
state_out[16] = state[1];
state_out[32] = state[2];
state_out[48] = state[3];
state_out[1]  = state[4];
// ...
```

S-Box Layer In the new, bit-wise representation, the S-Box can no longer easily implemented as a LUT[5]. Instead, we represent the S-Box output bits as Boolean functions in the input bits as explained in Section 2.1.3. In the following, we will mainly use the Algebraic Normal Form

[5] While one could of course still "collect" input bits, look-up the output in a LUT, and map the output bits back onto bytes, this would be quite inefficient.

representation (for details, cf. Section 2.3.1). For example, the ANF of the first output bit $y_0 \in \mathbb{F}_2$ of the PRESENT S-Box (with input $(x_0, x_1, x_2, x_3) \in \mathbb{F}_2^4$) is given as:

$$y_0 = x_0 + x_1 \cdot x_2 + x_2 + x_3$$

Hence, we can (again not in-place) implement the S-Box layer as follows:

```
// First S-Box, first bit
state_out[0]  = state[0] ^ (state[1] & state[2]) ^ state[2] ^ state[3];
...
// Second S-Box, first bit
state_out[4]  = state[4] ^ (state[5] & state[6]) ^ state[6] ^ state[7];
...
```

By writing the expression of the other S-Box output bits, we can hence implement the S-Box using Boolean operations only. The most important observation to understand the working principle of bitslicing is the following:

> All operations that we use (permuting bytes, applying Boolean operators) work *bitwise*, i.e., they operate on each bit independently. Hence, we can reclaim the "remaining" bits (so far set to zero) in each byte by storing a second block of plaintext in the second bit, in the third bit, and so on. We effectively execute the cipher *eight times in parallel*!

There are no further modifications needed to the above implementations of the blocks, the only requirement is to store the plaintext blocks p_i (each for example each with size 8 byte) in the correct way into the bitsliced state $s'[0...63]$ (with for example 64 bytes for 64 state bits): the first (least significant) bit of the first block p_0 goes into bit 0 of $s'[0]$, the second bit of p_0 into bit 0 of $s'[1]$, and so on. The first bit of the second block p_1 goes into bit 1 of $s'[0]$, the second bit of p_1 into bit 1 of $s'[1]$, and so on, until all 8 block are stored in s'. After the above operations (key addition, S-Box, permutation) have been carried out and the cipher is completely executed, one has to map back s' to a normal representation. The two representations (normal and bitsliced form) are illustrated in Figure 2.4.

Note that storing each entry of s' as one byte is a choice made here for illustrative purposes. Usually, one would select the register size of a processor (i.e., 16 bit for the MSP430, 32 bit for an embedded ARM core such as in the RP2040, 64 bit for a modern CPU) to be able to most efficiently make use of the processor's hardware. Of course, this implies that even more cipher instances are computed in parallel (16 for the example of the MSP430, 32 for the RP2040, 64 for a PC CPU).

Note: One actually does not have to explicitly transform the round keys into bitsliced form. Instead, if the same key (and hence the same round keys) are used for all bitsliced blocks, we can make use of the fact that each element (bit) `i` in the bitsliced `roundkey_exp[i]` can only either be all 0 (if bit `i` of the round key is 0) or all 1 (if bit `i` of the round key is 0). Hence, the key addition can be rewritten as (pseudocode):

```
for(uint8_t i = 0; i < 64; i++)
{
    if(bit i of roundkey == 1) {
```

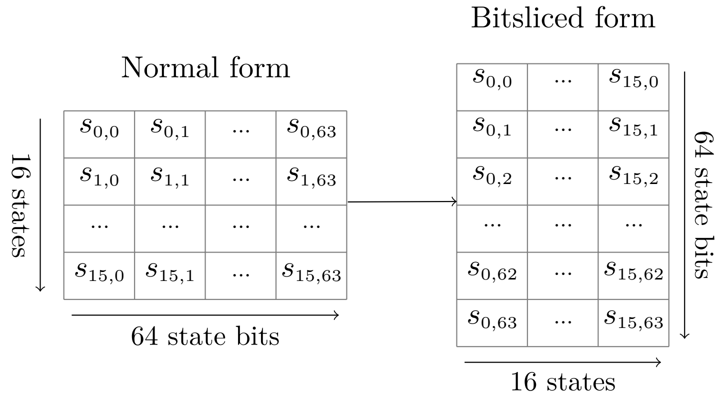


Figure 2.4: 16 cipher states (each 64 bit) in normal form (left) and bitsliced form (on a 16-bit processor, right).

```
        state[i] = state[i] ^ 0xFFFF...FF;
    }
}
```

However, this way causes the runtime of the implementation to be key-dependent, which is generally undesirable. Using the fact that -1 = `0xFFFF...FF` and that -0 = `0x0000...00`, we can however rewrite this as:

```
for(uint8_t i = 0; i < 64; i++)
{
    // Assume: x is either 1 or 0
    b = get bit i of roundkey;
    state[i] = state[i] ^ -b;
}
```

Time and Memory Complexity For bit-oriented ciphers like PRESENT or DES, bitslicing usually gives a significant performance increase due to the amount of operations saved for the permutations. The key addition layer is similar in performance for normal and bitsliced implementations, while the S-Box layer is usually more costly when implemented in bitsliced form. The overhead of the S-Box layer depends on the number of required Boolean operations to implement the S-Box—hence, finding a representation with as little Boolean operations as possible is crucial for bitsliced implementations with best performance. Besides, the operations to bring the input states into bitsliced form add computational complexity. Hence, it is normally desirable to stay in bitsliced form—and not to "switch" between the forms, e.g., to implement an S-Box as a LUT, converting the input to normal representation and back to bitsliced form afterwards. In terms of memory, bitslicing generates an overhead since multiple cipher states are stored at the same time. Moreover, the computation of the S-Box usually requires (some) temporary registers to buffer the input bits. The code size may increase as well, especially if the S-Box requires many Boolean operations. Overall, bitslicing is highly useful in scenarios where a large amount of data has to be encrypted with a bit-oriented cipher. In contrast, for byte-oriented ciphers like the AES (with larger S-Box), a table-based implementation will likely give better performance than bitslicing, though bitslicing can offer competitive performance and other advantages such as constant execution time (see Section 4.2 for why this is often needed) [SS17].

2.3.1 The Algebraic Normal Form

This section is supplementary material to the bitslicing section. We will look at one representation of Boolean functions, the Algebraic Normal Form, including an efficient algorithm to compute the ANF from a truth table for a given Boolean function. In this section, we consider a n-to-1 Boolean function:

$$y = f\left(x_0,\, x_1,\, \ldots,\, x_{n-1}\right) \text{ where } \left(x_0,\, x_1,\, \ldots,\, x_{n-1}\right) \in \mathbb{F}_2^n \text{ and } y \in \mathbb{F}_2$$

The ANF is a form to express any such function as the `XOR` sum of `AND` combinations of $x_0,\, x_1,\, \ldots$. As an example, for $n = 4$, a function in ANF would be:

$$f\left(x_0,\, x_1,\, x_2,\, x_3\right) = x_0 + x_1 + x_2 \cdot x_3 + x_0 \cdot x_1 \cdot x_3 + x_0 \cdot x_1 \cdot x_2 \cdot x_3$$

Note that ANF is written over binary variables in $\mathbb{F}_2^2$ (which is equivalent to computing $\mod 2$), hence, $+$ represents `XOR` and $\cdot$ `AND`. In general, the ANF can be expressed as:

$$f\left(x_0,\, x_1,\, \ldots,\, x_{n-1}\right) = \sum_{u \in \mathbb{F}_2^n} \left(\lambda_u \prod_{i=0}^{n-1} x_i^{u_i} \right)$$

with in total 2^n coefficients $\lambda_u \in \mathbb{F}_2$ (for $u = (00\ldots00)\,,\,(00\ldots01)\,,\,\ldots,\,(11\ldots11)$). Note that there is no “information loss” or “compression” in the ANF—both the truth table and the ANF can be described with a bit string of length 2^n. The ANF of any Boolean function can be computed using an Fast Fourier Transform (FFT)-like algorithm. The main steps are:

(1) Write the truth table for the function $y = f\left(x_0,\, x_1,\, \ldots,\, x_{n-1}\right)$;

(2) Apply the “butterfly” n times, with spacing $\sigma = 2^0 = 1,\, 2^1,\, 2^2,\, \ldots,\, 2^{n-1}$;

(3) The resulting table contains the coefficients λ_u, where $u = \left(x_0,\, x_1,\, \ldots,\, x_{n-1}\right)$.

whereas the butterfly (for two inputs a, b, spaced by σ) is defined as follows:

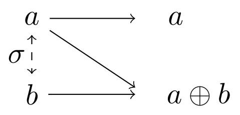


Figure 2.5: Butterfly for computing ANF.

x_2	x_1	x_0	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

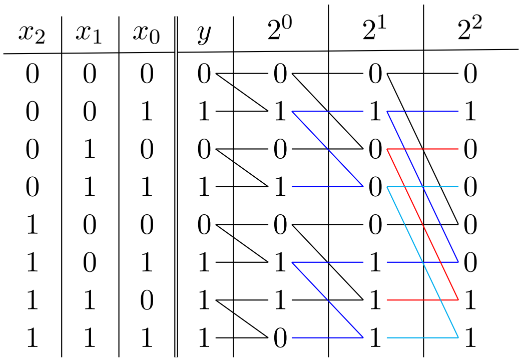
Figure 2.6: Truth table of example function $y = f(x_0, x_1, x_2)$.

Example The method is best illustrated by an example. Given the following function ($n = 3$): The ANF is computed as in the following Table 2.7:

After the final step with spacing $\sigma = 2^2 = 4$, one reads the coefficients that are set to 1, in this case these are the coefficients for input 001, 110, 111. This means that $\lambda_{001} = \lambda_{110} = \lambda_{111} = 1$, while the remaining λ_u are 0. Hence, we have the ANF:

$$y = f(x_0, x_1, x_2) = x_0 + x_2 \cdot x_1 + x_2 \cdot x_1 \cdot x_0$$

Note that if $\lambda_{000} = 1$, we would have the term $x_0^0 \cdot x_1^0 \cdot x_2^0 = 1$ in the ANF, i.e., the result is inverted (+1 is `XOR` with 1 or in other words inversion).



x_2	x_1	x_0	y	2^0	2^1	2^2
0	0	0	0	0	0	0
0	0	1	1	1	1	1
0	1	0	0	0	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	0
1	0	1	1	1	1	0
1	1	0	1	1	1	1
1	1	1	1	0	1	1

Figure 2.7: Computation of ANF of example function $y = f(x_0, x_1, x_2)$.

Chapter 3

Implementation of Asymmetric (Public Key) Cryptography

In contrast to symmetric cryptography, asymmetric (i.e., public key) cryptosystems have significantly higher computational and storage requirements (normally due to the longer keys, which range from 1024–8192 bit for RSA or 160–512 bit for Elliptic Curve Cryptography (ECC)). For PCs, this performance penalty is usually negligible, unless large amounts of data are to be encrypted asymmetrically. For embedded systems, however, implementing asymmetric cryptography efficiently is a current area of research. Most approaches favor ECC (mainly due to shorter key length), therefore we give an outlook on this area in Section 3.4. However, due to its conceptional simplicity, we will focus on RSA in this chapter, even though it is not overly suited for constrained embedded devices. Still, the basic building blocks (long-number arithmetic) are identical to more suitable approaches like ECC, hence, many optimizations directly carry over to such algorithms. A good reference for long-number algorithms is [MvOV96].

3.1 The RSA System

RSA is a well known algorithm. In this section, we only briefly repeat the main steps and introduce some notation. Note that the use of RSA in real applications has many additional requirements (padding scheme, key generation, etc.) to be secure.

Hence, the definitions/algorithms given here are to be understood as educational and must not be implemented straight away in a real system!

Definition 1. *For ℓ-bit RSA, pick two secret primes p, q, each about $\ell/2$ in size. Let $n = p \cdot q$. Furthermore, pick a public exponent e. The RSA public key is the tuple (n, e) The corresponding RSA private key is the tuple (p, q, d) with $d = e^{-1} \mod \phi(n)$.*

Definition 2. *RSA encryption of plaintext x under a public key (n, e) is given as:*
$y = x^e \mod n$.
Similarly, RSA signature verification of a signature s is: $x = s^e \mod n$.

Definition 3. *RSA decryption of a ciphertext y with a private key (p, q, d) is given as:*
$x = y^d \mod n$.
Similarly, an RSA signature on a plaintext x is computed as: $s = x^d \mod n$.

3.2 Long-Number Arithmetic

Asymmetric algorithms like RSA and ECC require computations with numbers beyond the width w of a single CPU register. Thus, we have to deal with *long-number* or *multi-precision arithmetic*. For PC-like platforms, there exist very comprehensive libraries to implement computations with long numbers (e.g., `GMP` or within `OpenSSL`). Hence, the need to implement such algorithms "from scratch" is not given; and for security and efficiency reasons it is not recommended to create own implementations.

However, for embedded applications, libraries developed for PC platforms may be oversized, contain many features not required in a particular case, or be inefficient due to assumptions that only hold for a PC architecture. On the other hand, it has educational value to study what happens "under the hood" of a long-number arithmetic library. Therefore, we will study a few building blocks of long-number arithmetic in the following subsections.

3.2.1 Number Representation

As mentioned, we assume a processor with w-bit registers. Given an ℓ-bit number u, we store this number in

$$k = \left\lceil \frac{\ell}{w} \right\rceil$$

registers. In other words, we represent the number u to the base $b = 2^w$:

$$\begin{aligned} u &= u_0 \cdot b^0 + u_1 \cdot b^1 + u_2 \cdot b^2 + \ldots + u_{k-1} \cdot b^{k-1} \\ &= u_0 \cdot 2^0 + u_1 \cdot 2^w + u_2 \cdot 2^{2w} + \ldots + u_{k-1} \cdot 2^{(k-1)\cdot w} \end{aligned}$$

where $u_0, u_1, \ldots, u_{m-1}$ are w-bit words often also called *limbs*. Note that each limb fits into a single w-bit register and can hence be handled by the CPU. Let us take the number $a = (12345678)_{10}$ as an example. In binary, this $\ell = 24$ bit number[1] is given as:

$$u = (101111000110000101001110)_2$$

Assume our CPU works (for some obscure reason) with $w = 5$ bit registers. Then, we would divide u into $k = \left\lceil \frac{24}{5} \right\rceil = 5$ limbs of 5 bit each:

$$u = (01011\ 11000\ 11000\ 01010\ 01110)_{2^5} = (a_4,\, a_3,\, a_2,\, a_1,\, a_0)_{2^5}$$

Note that we had to add a zero bit in the most-significant (leftmost) word so that the numbers fits into a multiple of the register width w.

The central problem of long-number arithmetic is now to express the standard operators (addition, subtraction, multiplication, modulo, division, ...) in terms of the above representation. For processors, the base is always a power of two (i.e., $b = 2^w$), but we occasionally may also use $b = 10$ for explanations (since this is the base most humans are used to).

We will mostly look at the most straightforward ("schoolbook") algorithm, but in some cases also consider optimized methods. Please note that in the following, we will denote the length of a number u *in base* b as k, i.e., $k = \log_b u$.

[1] The length of a number in bit can always be computed as $\ell = \lceil \log_2 u \rceil$.

3.2.2 Addition

For long-number addition, recall “addition with carry” used for computing larger sums manually for base $b = 10$. An example is shown in the following table:

u_i		9	6	3	4	5	6
$+ \; v_i$		9	4	8	0	2	4
carry c_i	1	1	1	0	0	1	
sum r_i	1	9	1	1	4	8	0

Table 3.1: Example: schoolbook addition with carry

Before we formalize this algorithm, we make a few short observations. First, note that in the first limb/digit addition $u_0 + v_0$, since u_i, v_i can take the maximum value $b - 1$ each, the largest possible result is $2b - 2$. Hence, the carry c (note that c does *not* mean ciphertext in this section) can at most take the value 1 (since the result is always $< 2b$). In the following limb additions $u_i + v_i + c_i$, the maximum result is $2b - 1$, i.e., still always $< 2b$. Finally, note that the sum of two k-digit numbers can be at most $k + 1$ digits long (if there is a carry from the most-significant limb addition $u_{\ell-1} + v_{\ell-1} + c_{k-1}$).

Algorithm 2 Long-number addition of two k-limb numbers u, v

```
$c \leftarrow 0$
**for** i = 0 ... $k - 1$ **do**
    $t \leftarrow u_i + v_i + c$
    $r_i \leftarrow t \mod b$
    $c \leftarrow \lfloor \frac{t}{b} \rfloor$
**end for**
$r_k \leftarrow c$
```

A couple of remarks: In Algorithm 2, the modulo and division operation do not correspond to real arithmetic instructions. Instead, $t \mod b$ means “take the lower limb of the result t”, and $\lfloor \frac{t}{b} \rfloor$ “take the upper limb of the result t”. These operations are hence easy to do for a processor (bit shifting and masking), and (incidentally) also for humans (read part of a number).

Note that t has to be a double-precision (two limbs) number for the algorithm to work, which is a slight overhead (because c is either 0 or 1). Many CPU architectures hence provide an “add-with-carry” (`ADDC`) instruction, which takes the input carry, performs the addition, and gives the output carry. However, there is no (portable) way to use add-with-carry in higher-level languages like `C` or `Java`. A workaround is to use Boolean operators to compute the carry without leaving single precision (single-limb computations). This can be achieved as follows (using the example of a 16-bit processor) [Bod]:

```
uint16_t r = u + v;
uint16_t carry =
    (((u & v) | (u & ~v & ~r) | (~u & v & ~r)) >> 15) & 0x1;
```

This code works since it sets the carry if (a) the topmost bit of both operands is set or (b) the topmost bit in the result and one operand is not set, but set in the remaining operand.

Further note that an alternative algorithm could be written with `if`-conditions: `if` $t \geq b$: $c \leftarrow 1$ `else` $c \leftarrow 0$. However, this would result in the execution time of the addition being dependent on the inputs, which is not desirable for a variety of reasons (some explained in Chapter 4).

Finally, subtraction can be done in a similar manner (carry becomes “borrow” bit), or by working in a two’s complement representation (a negative sign is expressed by negating each bit and adding 1).

Complexity Addition requires k base-b additions-with-carry, or $2k$ such additions if no `ADDC` is available. Its complexity is hence linear in k, i.e., $\mathcal{O}(k)$. It is hence generally computationally cheap, e.g., for adding two 2048-bit numbers on a 16-bit CPU, 128 limb-additions (with carry) are needed.

3.2.3 Multiplication

Similar to addition, we will devise an algorithm for long-number multiplication based on the method commonly taught in school. Before that, note that the product of a k-limb number and a m-limb number has length at most $k + m$ since

$$\left(b^k - 1\right) \cdot (b^m - 1) = b^{k+m} - b^k - b^m + 1 < b^{k+m}$$

Let us now compute the product of $u = 134$ and $v = 56$:

1	3	4	×	5	6
				2	4
	+		1	8	(0)
	+		6	(0	0)
			1		
	=		8	0	4
	+		2	0	(0)
	+	1	5	(0	0)
	+	5	(0	0	0)
		1			
	=	7	5	0	4

Table 3.2: Example: schoolbook multiplication

In principle, we have to multiply each digit of the second operand with each digit of the first operand and add up the results (with appropriate shifts by multiple of the base). The multiplication algorithm to compute $r = u \cdot v$ on a base-b processor is formalized in Algorithm 3.

$(y,\, x)_b$ is a “double limb” (base-b^2 digit), i.e., if a limb was represented with `uint16_t`, $(y,\, x)_b$ would be `uint32_t`. Note that $(y,\, x)_b$ has always of at most two base-b digits, since the maximum value is:

$$(b-1) \cdot (b-1) + (b-1) + (b-1) = b^2 - 2b + 1 + 2b - 2 = b^2 - 1 < b^2$$

Algorithm 3 Long-number multiplication of a k-limb numbers u with an m-limb number v

```
for $i = 0 \dots k + m - 1$ do
    $r_i \leftarrow 0$
end for
for $i = 0 \dots m - 1$ do
    $c \leftarrow 0$
    for $j = 0 \dots k - 1$ do
        $(y, x)_b \leftarrow r_{i+j} + u_j \cdot v_i + c$
        $r_{i+j} \leftarrow x$
        $c \leftarrow y$
    end for
    $r_{i+k} \leftarrow c$
end for
```

Complexity Algorithm 3 is a one-to-one realization of the "schoolbook" method, whereas the base-b shifts are implicit in the indices. It requires $k + m$ base-b multiplications, and $2\,(b + m)$ base-b^2 additions. If $k = m$, the complexity is hence quadratic in k, i.e., $\mathcal{O}(k^2)$. In the following we assume $k = m$ and examine if quadratic complexity is a lower bound for the complexity of multiplication (as it was assumed for a long time) or if we can improve the runtime.

3.2.4 Karatsuba Multiplication

For a long time, it was believed that $\mathcal{O}(k^2)$ is the best possible runtime for multiplication. However, in 1960, the Russian mathematician Anatoly Karatsuba proposed a method with lower runtime [KO63]. The algorithm is based on a simple divide-and-conquer approach, but was surprisingly discovered relatively recently (compared to other algorithms, which are sometimes known for centuries). After the discovery of the Karatsuba method, other, asymptotically even faster multiplication methods were found, e.g., the Toom-Cook and Schönhage-Strassen algorithms. However, these methods are usually only advantageous for bit lengths far beyond those used for asymmetric cryptography.

Karatsuba's algorithm is based on a simple observation. Split two k-digit numbers u, v into halves of equal size $\kappa = \left\lceil \frac{k}{2} \right\rceil$, i.e., write them as:

$$u = u_H \cdot b^\kappa + u_L$$
$$v = v_H \cdot b^\kappa + v_L$$

where u_H is the upper half $u_H = (u_{k-1} \dots u_\kappa)$ and u_L the lower $u_K = (u_{\kappa-1} \dots u_0)$ (similar for v). Now, writing the product $u \cdot v$ we get:

$$(u_H \cdot b^\kappa + u_L) \cdot (v_H \cdot b^\kappa + v_L) = u_H v_H b^{2\kappa} + (u_H v_L + u_L v_H)\, b^\kappa + u_L v_L$$

For computing the product in this way, we still need four half-length multiplications, which is the same as one full-length multiplication. However, Karatsuba's crucial observation was that the middle part $(u_H v_L + u_L v_H)$ can be expressed as:

$$(u_H + u_L) \cdot (v_L + v_H) - u_H v_H - u_L v_L = u_H v_L + u_L v_H$$

Since we already have to compute $u_H v_H$ and $u_L v_L$, we hence save *one* multiplication at the cost of two additions and two subtractions! In general form, the method is expressed with:

$$\begin{aligned} D_0 &= u_L \cdot v_L \\ D_2 &= u_H \cdot v_H \\ D_1 &= (u_H + u_L) \cdot (v_L + v_H) \end{aligned}$$

and hence:

$$u \cdot v = D_2 b^{2\kappa} + [D_1 - D_2 - D_0]\, b^{\kappa} + D_0$$

Example Let us see this with the example $1234 \cdot 4321$ in $b = 10$: Since $k = 4$, $\kappa = 2$. We thus have

$$u_H = 12,\ u_L = 34,\ v_H = 43,\ v_L = 21$$

and, computing the three products D_0, D_1, D_2:

$$\begin{aligned} D_0 &= 34 \cdot 21 = 714 \\ D_2 &= 12 \cdot 43 = 516 \\ D_1 &= (12 + 34) \cdot (43 + 21) = 46 \cdot 64 = 2944 \end{aligned}$$

Thus, $u \cdot v = 516 \cdot 10^4 + [2944 - 714 - 516]\, 10^2 + 714 = 5160000 + 171400 + 714 = 5332114$.

Recursion Note that the Karatsuba method can (and should) be applied *recursively*, i.e., to compute all the sub-products D_0, D_1, D_2 again by applying the method, and so on. In theory, the algorithm is applied until we only reach base-b operands. In the above example, we would have 1 recursion (2 iterations), after which we reach single-digit products. E.g., to compute the sub-product $34 \cdot 21$, we have (leaving out indices for simpler notation):

$$\begin{aligned} D_0 &= 4 \cdot 1 = 4 \\ D_2 &= 3 \cdot 2 = 6 \\ D_1 &= (4 + 3) \cdot (2 + 1) = 21 \end{aligned}$$

and thus $34 \cdot 21 = 6 \cdot 10^2 + [21 - 4 - 6]\, 10 + 4 = 600 + 110 + 4 = 714$.

In practice, however, the algorithm is usually only applied until an (experimentally determined) threshold is reached. Afterwards, the schoolbook algorithm is applied. The rationale for this is that at some point the additional additions and management operations become more expensive than the advantage of saving one multiplication.

Complexity We start with the case of one iteration ($i = 1$, no recursion). In this case, to compute a product of two k-digit numbers, we need three half-length multiplications, i.e., have complexity (ignoring additions):

$$3 \left(\frac{k}{2} \right)^2 = \frac{3}{4} k^2$$

Applying the method again recursively (iteration $i = 2$) to compute D_0, D_1, D_2, we hence have three products of $\frac{k}{2}$-digit numbers to compute. This has then complexity:

$$3\left(\frac{3}{4}\left(\frac{k}{2}\right)^2\right) = \frac{9}{16}k^2$$

The general complexity for Karatsuba with i iterations is hence

$$\mathcal{O}\left(\left(\frac{3}{4}\right)^i k^2\right)$$

Assume we multiply multiply two numbers with length $k = 2^i$ (i.e. need i iterations). Then, we can refactor this expression as follows:

$$\begin{aligned}\left(\frac{3}{4}\right)^i k^2 &= 3^i\left(\frac{1}{2^2}\right)^i\left(2^i\right)^2 = 3^i\frac{2^{2i}}{2^{2i}} = 3^i\\ &= 3^{\log_2 k} = \left(2^{\log_2 3}\right)^{\log_2 k} = \left(2^{\log_2 k}\right)^{\log_2 3}\\ &= k^{\log_2 3} \approx k^{1.585}\end{aligned}$$

$\mathcal{O}(k^{1.585})$ is the (quite famous) complexity estimation for Karatsuba. Note that the exponent $\log_2 3$ indicates the number of half-length multiplications in the recursion step. If we would not apply the Karatsuba "trick", we would have $\log_2 4$, which brings us back to the original complexity $\mathcal{O}(k^2)$ of the schoolbook method.

3.2.5 Modulo Arithmetic

Modulo arithmetic, i.e. computing the remainder $u \mod n$ for large u, n, is a problem related to long-number division: given the quotient $\gamma = \lfloor\frac{u}{n}\rfloor$, the remainder $r = u \mod n$ is given as $r = n - \gamma \cdot u$.

However, long-number division (see e.g. [MvOV96, 14.2.5]) is amongst the most complicated long-number algorithms. For example, when dividing a $2k$-number number by a k-digit number, $k \cdot (k-2)$ digit multiplications and k digit divisions are consumed.

There are, however, methods to reduce the computational complexity of modular arithmetic, including the Montgomery reduction [MvOV96, 14.3.2] and the Barrett reduction [MvOV96, 14.3.3]. Barrett reduction is used in the assignment for this lecture and is already provided in the respective template. The method is based on performing the (expensive) pre-computation

$$\mu = \left\lfloor\frac{b^{2k}}{n}\right\rfloor$$

which then allows to avoid digit divisions and essentially requires $\mathcal{O}(k^2)$ single-digit multiplications.

Further optimizations are possible for reductions modulo a number of "special form", e.g., for a number $n = b^t - \delta$, where δ is ideally a small number. Such a modulus is e.g. often encountered for ECC or normal Diffie-Hellman-based protocols [MvOV96, 14.3.4].

3.3 Exponentiation Algorithms

Exponentiation, i.e., computing $y = x^e \mod n$, for x, e, n long numbers, is a core operation in many cryptographic schemes, including RSA (see Def. 2 and Def. 3) and the Diffie-Hellman key exchange. In similar form (as "scalar multiplication"), it is also required for ECC.

There are many methods to efficiently implement exponentiation in software. We only focus on the most basic method, the left-to-right Square-and-Multiply (SAM) algorithm. Other techniques, including k-ary and sliding window exponentiation, allow to substantially reduce the runtime of exponentiation. However, they are beyond the scope of these lecture notes—the interested reader is referred to [MvOV96, 14.6].

The key idea of the SAM algorithm is to interleave the modular reduction and multiplications. Otherwise, if performing the exponentiation first and the reducing $\mod n$, the storage and computational requirements would be immense (take into account that multiplying two k-digit numbers results in a $2k$-digit product).

The algorithm hence "scans" the binary exponent e left-to-right (i.e. starting at the Most Significant Bit (MSB)), performs a squaring for each bit (irrespective of the bit being 0or 1) and an additional multiplication if the respective bit is set to 1. Let $e = (e_{t-1}\ e_{t-2}\ \dots\ e_0)_2$ be the exponent in binary form. Assume that $e_{t-1} = 1$, i.e. the MSB is always set. x, n are k-digit numbers. The SAM algorithm to compute $y = x^e \mod n$ is then formalized in Alg. 4:

Algorithm 4 Left-to-right square-and-multiply algorithm to compute $y = x^e \mod n$

```
$y \leftarrow x$
for i $= t-2$ ... $0$ do
    $y \leftarrow y^2 \mod n$
    if $e_i = 1$ then
        $y \leftarrow y \cdot x \mod n$
    end if
end for
```

Example Let us look at an example for the exponent $e = 11011_2$. We have $t = 5$. The following table shows the operation and the current state of the exponent for the SAM:

i	Current exponent	Current y	Operation
–	1_2	x	LOAD
3	10_2	$x^2 = x^{10_2}$	SQ
3	11_2	$x^{10_2} \cdot x = x^{11_2}$	MUL
2	110_2	$\left(x^{11_2}\right)^2 = x^{110_2}$	SQ
1	1100_2	$\left(x^{110_2}\right)^2 = x^{1100_2}$	SQ
1	1101_2	$x^{1100_2} \cdot x = x^{1101_2}$	MUL
0	11010_2	$\left(x^{1101_2}\right)^2 = x^{11010_2}$	SQ
0	11011_2	$x^{11010_2} \cdot x = x^{11011_2}$	MUL

Table 3.3: Example: SAM

Complexity The `for` loop is executed $t-1$ times. We hence need $t-1$ squarings and modular reductions (`SQ`). If e_i is set, we in addition execute a multiplication and modular reduction (`MUL`). On average, assume that half of the bits of e are set. Then, we need

$$\frac{t-1}{2}\text{ MUL and }t-1\text{ SQ}$$

3.4 Outlook: Elliptic Curve Cryptography

In contrast to RSA, where key sizes over 2048 bit are generally recommended today, Elliptic Curve Cryptography reaches a similar security level with shorter keys. For example, to reach the equivalent security level to RSA-3072, 256-bit ECC is sufficient. This means that the required memory for storing public and private keys is reduced from 384 byte to 32 byte, i.e., by a factor of 12. Apart from that, the long-number routines become faster due to the shorter operands. In contrast, however, a single group operation (the equivalent to a multiplication $\mod n$ for RSA) requires multiple long-number operations.

Still, overall, the other advantages make ECC better suited for constrained embedded devices. Optimized implementations, e.g., based on Curve25519 [Ber06], can achieve runtimes under 1 s for a full ECC operation (signature, encryption, etc.) on an MSP430 [HMH+14]. Public domain implementations of Curve25519 for common µCs are available at: `http://munacl.cryptojedi.org/`. A general overview over ECC can e.g. found in [CFA+12], which is available online at: `http://cs.ucsb.edu/~koc/ccs130h/2013/EllipticHyperelliptic-CohenFrey.pdf`.

Chapter 4

Implementation Attacks

For a long time, it was believed that designing a mathematically secure algorithm is sufficient to protect the respective cryptographic keys. The cryptographic algorithm was regarded as a black box, i.e., the adversary only knows inputs, outputs, the internal structure, but does not have access to internal, intermediate values. Around 1997 (and in governmental agencies probably much earlier), it was found that this notion is insufficient if the adversary has *physical access* to the cryptographic Device Under Test (DUT). Techniques like active Fault Injection (FI) and passive Side-Channel Analysis (SCA) were shown to be able to break analytically secure ciphers like DES, AES, and RSA in minutes or seconds [KJJ99, BDL97, Koc96].

The field of *implementation attacks* comprises a large number of different attack techniques. What all methods have in common is that they exploit properties of the actual implementation, not the mathematical assumptions. For example, for software implementations, *buffer overflows* are an implementation attack: Even if the implemented cipher is secure, a buffer overflow that allows to directly read cryptographic keys renders the security assumptions invalid.

In the field of embedded systems and hardware, implementation attacks are often based on measuring or manipulating physical properties, e.g., recording the power consumption, inducing faults during computations, and so on. We will focus on these two types of implementation attacks. However, note that other classes (e.g., software vulnerabilities like buffer overflows) are equally important for embedded systems.

4.1 Fault Injection

Every computing device (and especially µCs) requires certain conditions to function normally and compute correct results. These include:

- A stable supply voltage (typically 3.3 V, 1.8 V, or 5 V in older systems),
- a stable clock signal (often supplied by an external oscillator),
- “normal” operating temperature (usually between -15° C and 80° C),
- no strong magnetic or electric fields in the vicinity, and many more conditions.

If one or more of these conditions are not met, the device may start to return incorrect results. For instance, memory reads or writes could fail, arithmetic operations return incorrect results, conditional jumps not be executed, and so on. An adversary can hence induce such *faults* on purpose, for example, with one of the following methods:

- Expose the device to high or low temperatures. Since temperature changes slowly, this will have effects for longer amounts of time. It could for example reduce the entropy of a Random Number Generator (RNG) or disable memory writes.
- Manipulate the supply voltage to expose the device to short overvoltages (“pulses”) and undervoltages (“glitches”). If precisely timed, this can affect a single or a few instructions.
- Temporarily increase the clock frequency (or change the clock signal shape) to overclock the device for a short period. Similar to the supply voltage, this can have effects on the instruction level.
- Open the Integrated Circuit (IC) package and and expose the circuit to Ultraviolet-C (UV-C) light. This can for example clear Flash and EEPROM memory. Aternatively, a photo flash can have similar effects.
- Open the IC package and target specific parts of the circuitry with a focused laser beam. This allows modifications of single signals within the target device (e.g., a single memory location or a single bit of a data bus) with sub-instruction precision. However, the necessary equipment is much more expensive than for the above techniques.
- Use so-called microprobes to tap and change internal signals on the opened IC. For example, cryptographic keys could just be read from memory, or internal signals be changed precisely. Another possibility is the use of a so-called Focused Ion Beam (FIB) to change the IC structures on a microscopic level. However, this requires extremely expensive equipment usually only found in high-end semiconductor test labs.

Note that the above list is by far not exhaustive. A good overview is given in [HCN+06].

4.1.1 Generic FI Attacks

There are certain FI attacks that are general, i.e., apply to every cipher. An easy-to-grasp example is the skipping of an operation. Take the following example of a PIN comparison:

```
int c = memcmp(entered_pin, stored_pin, pin_len);

if(c != 0)
{
    return -1;
}

// Code continues ...
```

Assume a fault can be induced with instruction-level precision, then, the adversary can simply skip the “return -1;” in the `if` condition and bypass the PIN check. Similarly, it could be possible to bypass a signature verification, a MAC check, and so on.

A different generic attack [BS97] assumes the following *fault model*: a fault induced on a single bit always sets the bit to zero, i.e., a 1 is turned into a 0, a 0 stays at 0 (“stuck-at” fault). Furthermore assume that the adversary can selectively fault each single bit of cryptographic

Algorithm 5 General fault attack with “stuck-at” fault model

```
Pick any plaintext $p$
$c_{ref} \leftarrow e_k(p)$
for $i = 0 \dots m-1$ do
    $c \leftarrow \overline{e}_k^{(i)}(p)$
    if $c = c_{ref}$ then
        Key bit $i$ is 0
    else
        Key bit $i$ is 1
    end if
end for
```

key during the load. Let e_k be a cipher with m-bit key k, and $\overline{e}_k^{(i)}$ indicate the cipher where a fault has been induced for the ith key bit. Then, the adversary recovers the full key as follows:

The attack of Algorithm 5 works because the adversary can see if faulting a single bit has an effect on the ciphertext. If the ciphertext changed compared to the correct one c_{ref}, he concludes that the key bit was 1 (fault changed a 1 to a 0). Otherwise, the fault left a 0 at 0, and the ciphertext is unchanged.

Both generic attacks require significant control over the fault and its location in time and “space”. In the following subsection, we look at attacks tailored to specific ciphers, hence requiring less control over the fault parameters.

4.1.2 FI on CRT-RSA

The attacks presented in this section are amongst the earliest reported fault attacks [BDL97]. The attack is commonly to referred as “Bellcore” attack (due to the affiliation of the authors), and an improvement by Arjen Lenstra as the “Lenstra” attack. Both target RSA with a very common optimization based on the Chinese Remainder Theorem (CRT). Before we describe the attack, we briefly explain this optimization.

RSA with CRT The Chinese Remainder Theorem allows to split one RSA computation modulo the k-digit number n (e.g., signature generation $s = x^d \mod n$) into two operations modulo p and q. Both p and q have approximately half the bit length of n, i.e., are $k/2$ digits long. The algorithm to compute $s = x^d \mod n$ (or equally decryption) works as described in Algorithm 6.

To understand the advantage of this approach, recall from Section 3.2.3 that multiplication has complexity $\mathcal{O}(k^2)$, and that the SAM algorithm takes approximately $1.5\,\ell$ multiplications for an ℓ bit exponent. Hence, for the normal computation, we have $1.5\,\ell\,k^2$. For the CRT optimization, we need two modular exponentiations with half the bit length for operands and exponent. Hence, we have complexity

$$2 \cdot 1.5\,\frac{\ell}{2}\left(\frac{k}{2}\right)^2 = \frac{1.5}{4}\,\ell\,k^2$$

Algorithm 6 CRT-RSA signature computation

```
The following quantities are pre-computed once:
$d_p \leftarrow d \mod p-1$
$d_q \leftarrow d \mod q-1$
$c_p \leftarrow q^{-1} \mod p$
$c_q \leftarrow p^{-1} \mod q$
Then, compute:
$x_p \leftarrow x \mod p$
$x_q \leftarrow x \mod q$
$s_p \leftarrow x_p^{d_p} \mod p$
$s_q \leftarrow x_q^{d_q} \mod q$
Recombine result:
$s \leftarrow [q \cdot c_p] \cdot s_p + [p \cdot c_q] \cdot s_q \mod n$
```

In other words, the CRT allows us to reduce the computational complexity by a factor of *four*, neglecting the (minor) efforts for splitting the input and recombining the result. This is why it is very common especially in constrained embedded devices.

Bellcore Attack The fault model for the Bellcore attack is very relaxed: We assume that the adversary can inject any fault into the computation $s_p = x_p^{d_p}$ (or equivalently $s_q = x_q^{d_q}$). Then given a correct signature s and a faulty signature $\overline{s}$, the modulus n can be factored with high probability as follows:

$$q = \gcd\left(s - \overline{s},\, n\right),\; p = \frac{n}{q}$$

The reason why this works is given in the following equation:

$$s - \overline{s} = [q \cdot c_p] \cdot s_p + [p \cdot c_q] \cdot s_q - [q \cdot c_p] \cdot \overline{s_p} - [p \cdot c_q] \cdot s_q = q \cdot c_p \cdot \left(s_p - \overline{s_p}\right) \; = \lambda\, q$$

i.e., we get a multiple of q for which $\lambda \neq p$ almost always. Hence, $\gcd\left(\lambda\, q,\, p \cdot q\right) = q$.

Lenstra Attack The "disadvantage" of the Bellcore attack is that it requires a correct and a faulty signature on the same plaintext x. Thus, if the adversary cannot control the input to the signature, e.g., due to randomization, the attack does not work. However, Lenstra's improvement allows to apply the attack by "emulating" the correct value using signature verification. More precisely, assume we have a faulty signature $\overline{s}$ as above and the corresponding plaintext x. Then, note that (since the fault was induced on s_p), the result is still correct modulo q, i.e.,

$$\overline{s} \mod q = [p \cdot c_q] \cdot s_q = s \mod q$$

In other words, if we "verify" the signature modulo q, we have

$$\overline{s}^e \mod q = s^e \mod q = x \mod q$$

Thus, in general (not modulo q), we have

$$\overline{s}^e = x + \delta \cdot q \;\Leftrightarrow\; \overline{s}^e - x = \delta \cdot q$$

with $\delta \neq p$ with high probability, i.e., again a multiple of q. Thus, we can again factor:

$$q = \gcd\left(\overline{s}^e - x,\, n\right),\, p = \frac{n}{q}$$

4.1.3 Countermeasures

Countermeasures against fault attacks can be generally divided into two classes: general *detection-based* and *algorithmic* approaches. Detection-based countermeasures are often implemented on the hardware or a low software level and aim to detect that FI is taking or has taken place, independent of the specific algorithm. Examples for this include:

- Monitoring of power supply and clock signal for glitches,
- monitoring of environmental conditions like temperature,
- use of sensors to detect light and laser FI,
- redundant computations, e.g., with two identical CPUs in parallel or two times subsequently,
- insertion of checks into program code to ensure that the control flow is not manipulated, and
- checksums and partity bits on internal signals and the data bus.

In contrast, algorithmic countermeasures make use of certain properties of a specific algorithm (e.g., RSA or AES) to detect faults. These include:

- "Infective computations" [STA+10] to render faulty results useless,
- computation of an algorithm and its inverse, e.g., directly verifying an RSA signature or decrypting an AES ciphertext before outputting it,
- additional, more efficient checks like an approach patented by Shamir, see e.g. [Joy09], and
- insertion of dummy operations to make it harder to inject a fault at a specific position.

Note that the above lists are by far not complete—numerous research papers have been published in the area of FI countermeasures since around 1997. The question arises how an embedded device should react on detection of a potential fault. This of course depends very much on the application and the required level of security. For instance, a high-security smartcard may completely seize to function after a relatively low number of faults have been detected. On the other hand, a device with high reliability requirements (e.g., an automotive Electronic Control Unit (ECU)) may continue to work even if potentially intentional faults occur, but for example block the access to cryptographic operations for some time or at least log the incident.

A cautionary note on counters for counting FI attempts: first of all, such counters require some form of permanent memory (e.g., Flash or EEPROM), that is not always available. Secondly, writes to such memories can be usually observed in the power consumption or other side channels (see Section 4.2). Hence, care must be taken how such counters are incremented. The most obvious approach of incrementing the counter after the detection of FI enables the following attack:

An adversary injects a fault, and observes whether a memory write is about to occur (i.e., the counter is incremented). If this is the case, he removes the power from the target device to prevent the memory write. Then, he slightly adjusts his FI parameters (e.g., aim laser at a different location, reduce amount of voltage glitch or pulse, change moment in time, etc), and attempts a new FI. This renders the use of counters ineffective alltogether.

The correct way for using fault counters would be: On the start of a cryptographic operation to be protected, increment and write the counter. Then, carry out the operation. Only if the operation was successful, decrement the counter again. The main disadvantage is that for *every* invocation of the protected operation, a memory write is now required, which is expensive in terms of power consumption and computation time. In addition, permanent memories like Flash and EEPROM have limited write cycles (in the range of 10,000 . . . 1,000,000 cycles). However, new memory technologies (like FRAM used in the MSP430 variant used for this lecture) may help to solve this problem, since they offer permanent storage at little additional cost and have virtually unlimited write cycles.

4.2 Side-Channel Analysis

Side-Channel Analysis (SCA) is based on the passive measurement of physical properties of an implementation while it performs cryptographic operations. The side-channel signal or leakage is usually called *trace*. Examples for some of the most common side channels include:

Execution time The execution time or "timing" may leak information on secret data, for example through different code branches [Koc96] being taken or through cache-timing side channels [Ber].

Power consumption The power (or rather current) consumption of an IC depends on the processed data. This is probably the most classical side channel and was introduced by Kocher et al. around 1996 [KJJ99].

Electro-Magnetic (EM) emanation The EM emanation [AARR03] of an IC can give information equivalent to the power consumption, but also allow very detailed, localized measurements of single circuit components.

Photonic emissions The photonic emissions (recorded with a microscope and an infrared-sensitive camera) of an opened IC can yield a very precise picture of the internal processes, down to the bit level [FH08, SNK$^+$12].

Sound The high-frequency sound due to vibrating circuit components can leak information especially for slow algorithms like RSA [ST04].

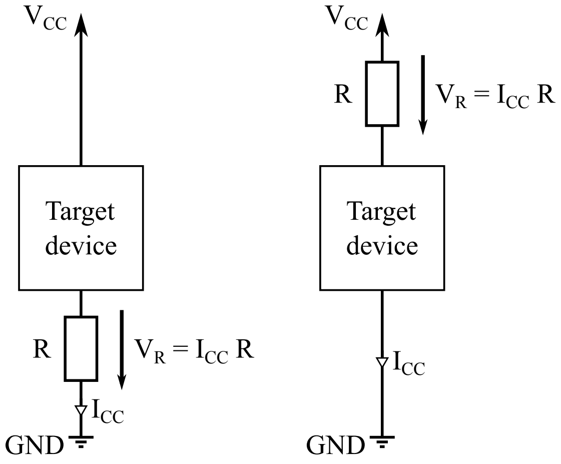


Figure 4.1: Typical measurement setup for power consumption side-channel attacks

Measurement Setup Figure 4.1 shows the two most typical setups for measuring the power (current) consumption of a target device. In the left setup, a (small) shunt resistor R^1 is inserted into the ground path of the target device. Since the supply current I_{CC} flows through the resistor, it can be derived by measuring the voltage drop V_R over R as $I_{CC} = \frac{V_R}{R}$. Note that in SCA, usually, we are not interested in the absolute value of I_{CC}, but rather require only a value proportional to it. Hence, in practice, one can just measure V_R with a Digital Storage Oscilloscope (DSO) and ignore the conversion factor R.

An alternative approach is to place R into the V_{CC} path and again measure the drop over V_R. Since I_{CC} is the same for both paths, this approach is equivalent to measuring in the ground path (if there is only one supply voltage). However, when attaching a standard DSO probe to measure V_R in the V_{CC} path, note that one obtains $V_{CC} - V_R$, i.e., the signal is DC-shifted. Hence, one has to remove the constant DC component by measuring AC-coupled. Alternatively, V_R can be directly measured using a so-called differential probe.

4.2.1 Simple Power Analysis

Simple Power Analysis (SPA) is an umbrella term for side-channel attacks that work by (visually) inspecting one or a few traces. For instance, take the SAM (Algorithm 4): if an adversary can distinguish squaring (SQ) and multiply (MUL), he can trivially reconstruct the secret exponent bit-by-bit. Take the example of the trace in Figure 4.2: The executed operation sequence is: SQ, MUL, SQ, MUL, SQ, SQ, SQ, MUL, SQ, SQ, MUL.

4.2.2 Differential Power Analysis

While SPA attacks utilize larger amounts of side-channel leakage (e.g., of a long-number multiplication consuming many cycles), Differential Power Analysis (DPA) [KJJ99] can exploit tiny

[1] usually between 1 Ω and several hundred Ω

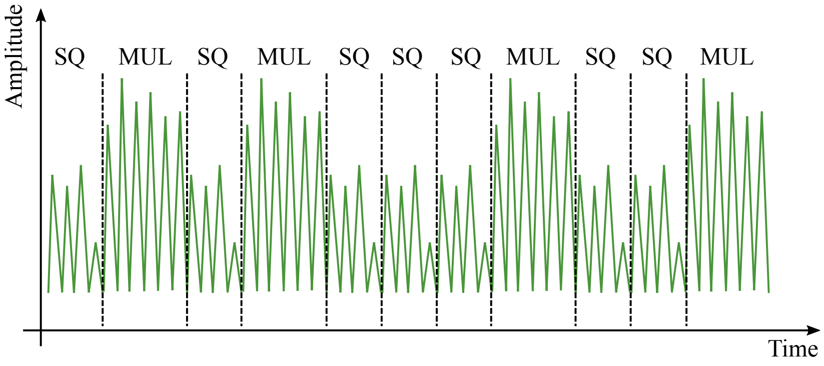


Figure 4.2: SPA of SAM algorithm for RSA signature

leakages (e.g., of a single bit being processed on a large IC). To this end, DPA uses statistical methods to detect the tiny leakage signal in a larger amount of (noisy) traces. The core assumption of DPA is that the power consumption (or other side-channel signals) are slightly different if the target device processes a zero bit or a one bit. However, in contrast to SPA, this difference does not have to be "visible" and further can be hidden in noise.

A typical DPA is divided into two steps, the measurement and evaluation phase, which are described in the following.

Measurement DPA operates on a set of n traces $p_i(t)$ with T sample points each (i.e., $t = 0 \ldots T-1$). Each trace is the power consumption while the target device encrypts a plaintext X_i (or performs another cryptographic operation) using the key k. In the following, when we talk about bytes (or other parts) of x_i, we will use the notation X_i^0 to for example denote byte 0 of X_i. If it is clear from the context that we only talk about a specific byte, we write x_i. The measurement phase is summarized in Algorithm 7.

Algorithm 7 Measurement phase of a DPA

```
for $i = 0 \ldots n-1$ do
    Generate random, uniformly distributed plaintext $X_i$
    Send $X_i$ to device, device computes $enc_K(X_i)$
    Measure power consumption $p_i(t)$
    Store $p_i(t)$ and corresponding $X_i$
end for
```

Evaluation The evaluation process (for the first byte/part k of the round key) for the example of the AES is summarized in Algorithm 8.

Why DPA works DPA apparently allows to target the leakage of a single bit of a single register or memory location in a potentially very large circuit. The question arises why this approach works. The main reason is that we *average* many signals, which reduces the amount of noise

Algorithm 8 Evaluation phase of a DPA for first byte of AES

```
$S_0^{\hat{k}}(t) \leftarrow$ empty set for each candidate $\hat{k}$ for $k$
$S_1^{\hat{k}}(t) \leftarrow$ empty set for each candidate $\hat{k}$ for $k$
for $i = 0 \ldots n-1$ do
    Load trace $p_i(t)$ and plaintext $X_i$
    $x_i \leftarrow$ first byte/part of $X_i$
    for $\hat{k} = 0 \ldots 255$ do
        $b \leftarrow S\left(x_i \oplus \hat{k}\right)$
        if $\mathrm{LSBit}(b) = 0$ then
            Add $p_i(t)$ to $S_0^{\hat{k}}$
        else
            Add $p_i(t)$ to $S_1^{\hat{k}}$
        end if
    end for
end for
for $\hat{k} = 0 \ldots 255$ do
    $\overline{S}_0^{\hat{k}}(t) \leftarrow \frac{1}{\left|S_0^{\hat{k}}\right|} \sum_{p_i \in S_0^{\hat{k}}} p_i(t)$
    $\overline{S}_1^{\hat{k}}(t) \leftarrow \frac{1}{\left|S_1^{\hat{k}}\right|} \sum_{p_i \in S_1^{\hat{k}}} p_i(t)$
    $\mathrm{DPA}^{\hat{k}}(t) \leftarrow \overline{S}_1^{\hat{k}}(t) - \overline{S}_0^{\hat{k}}(t)$
end for
Find $\mathrm{DPA}^{\hat{k}}$ with highest peak to recover $k = \hat{k}$.
```

relatively to the signal (i.e., the leakage of a single bit). More precisely, we can write a power trace at a point in time t_0 as

$$p(t_0) = p_{signal} + \mathcal{N}_{alg} + \mathcal{N}_{measure}$$

where p_{signal} is the actual leakage, $\mathcal{N}_{alg}$ algorithmic noise (caused by all the parts of the circuit we do not predict), and $\mathcal{N}_{measure}$ measurement noise (i.e., physical effects in the setup etc.). Assuming $\mathcal{N}_{alg}$, $\mathcal{N}_{measure}$ are Gaussian, for $n \to \infty$, the noise vanishes and we only see the differences between p_{signal} (bit = 0 and bit = 1) in the difference of means. In practice, $n \to \infty$ can mean anything between a few ten to millions or billions of traces.

4.2.3 Correlation Power Analysis

In a DPA, we focus on a single bit, while we actually predict more information (e.g., 8 bit for the AES). To make use of this fact and reduce the number of required traces, Correlation Power Analysis (CPA) [BCO04] makes use of a *leakage model*—i.e., the fact that we can approximate how an internal value b affects the leakage signal p_{signal}. Figure 4.3 shows an example how a single byte could leak: internal values with a lower number of ones result in a lower amplitude, while more ones lead to higher values.

This leakage model is the Hamming Weight (HW) model: the side-channel signal is proportional to the HW of the processed value, i.e., we have $p_{signal} \propto \mathrm{HW}(b)$. As a reminder, $\mathrm{HW}(b)$ is defined as the number of bits set to 1 in b. For instance, $\mathrm{HW}(0x00) = 0$, $\mathrm{HW}(0xAB) = \mathrm{HW}(10101011) = 5$, and $\mathrm{HW}(0xFF) = 8$.

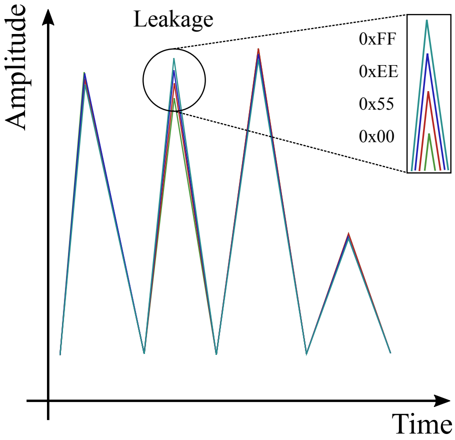


Figure 4.3: Leakage of a single byte value b

There are many other conceivable leakage models, the most prominent one being the Hamming Distance (HD) model, that assumes that the leakage depends on the *previous* value b' and the *new* value b of a register. The HD is the number of bits that have changed between b' and b, i.e., $\mathrm{HD}(b', b) = \mathrm{HW}(b' \oplus b)$.

In the following, we use notation based on Section 4.2.2 to explain how CPA allows to include the leakage model into an approach similar to DPA. Again, we have n traces $p_i(t)$ with T sample points each and the corresponding plaintexts X_i. Again, take the example of the AES and assume a key candidate $\hat{k}$ for the first S-Box with input byte x_i. Then, we predict the intermediate value as:

$$b_{i,\hat{k}} \leftarrow S\left(x_i \oplus \hat{k}\right)$$

Now, we employ our leakage model $f(\cdot)$ to convert this value into a hypothetical power consumption (e.g., using the HW model):

$$h_{i,\hat{k}} = f\left(b_{i,\hat{k}}\right) \overset{\text{e.g.}}{=} \mathrm{HW}\left(b_{i,\hat{k}}\right)$$

The final step is to determine the "match" between the predictions $h_{i,\hat{k}}$ for a key candidate and the reality of the traces $p_i(t)$. To this end, we use *Pearson's correlation coefficient* ρ, cf. [Wik16c]. ρ estimates the linear correlation between two data sets A_i and B_i: if they are perfectly correlated, $\rho = 1$ and one data set can be written as a linear function $A_i = \alpha \cdot B_i + \beta$ of the other. If there is no correlation, $\rho = 0$. For perfectly inverse-correlated data sets, $\rho = -1$. In summary, keep in mind that $-1 \leq \rho \leq 1$.

In our case, one data series is the set of traces $p_i(t)$, the other one the prediction $h_{i,\hat{k}}$. Since $p_i(t)$, we compute the correlation point-wise for each time sample. However, note that $h_{i,\hat{k}}$ is not time-dependent, hence, this data series is always the same for all t.

Thus, given a key candidate $\hat{k}$, we obtain:

$$\rho_{\hat{k}}(t) = \frac{\sum\limits_{i=0}^{n-1} \left(h_{i,\hat{k}} - \overline{h}_{\hat{k}}\right) \cdot \left(p_i(t) - \overline{p}(t)\right)}{\sqrt{\sum\limits_{i=0}^{n-1} \left(h_{i,\hat{k}} - \overline{h}_{\hat{k}}\right)^2 \cdot \sum\limits_{i=0}^{n-1} \left(p_i(t) - \overline{p}(t)\right)^2}} = \frac{\operatorname{cov}\left(h_{i,\hat{k}}, p_i(t)\right)}{\sqrt{\operatorname{var}\left(h_{i,\hat{k}}\right) \cdot \operatorname{var}\left(p_i(t)\right)}}$$

$\rho_{\hat{k}}(t)$ is the equivalent to the difference-of-means curve $\mathrm{DPA}^{\hat{k}}(t)$ in a DPA—the correlation with the highest peak identifies the correct key candidate.

Rules of Thumb In contrast to DPA, where the value of a peak in the difference-of-means is hard to predict, CPA is normalized to the range $-1 \leq \rho \leq 1$, which makes it easier to define what a *significant* correlation is. [MOP07] provides several "rules of thumb" for the correlation coefficient. First, for n traces, the expected "noise level" ρ is

$$\rho_{noise} = \frac{4}{\sqrt{n}}$$

That means that any correlation $\leq \rho_{noise}$ is essentially meaningless and not significant. This formula reflects the fact that for more traces, it is possible to identify smaller correlations. The second rule of thumb allows to estimate the minimum number of required traces n_{min} to unambiguously extract a key (with very high probability) when the respective correlation for the correct key ρ_{key} is known (e.g., from prior experiments).

$$n_{min} = \frac{28}{\rho_{key}^2}$$

This formula only holds for $\rho_{key} \leq 0.2$. To give an example: Assume after $n = 2000$ traces, we get $\rho_{key} = 0.1$. Clearly, this correlation is significant since $\rho_{noise} = {}^{4}/_{\sqrt{2000}} = 0.089$. However, to be able to unambiguously extract the key with that correlation, $n_{min} = {}^{28}/_{0.1^2} = 2800$ traces would be required.

4.3 Countermeasures

Protecting against SCA has been an active area of research since the initial discovery around 2000. Hence, we can only look at a small selection of countermeasures in this section. A deeper survey for block ciphers is for example given in [Pro13]. Countermeasures to directly thwart SCA can be realized on two dimensions (and a combination of them): the amplitude of the leakage and the timing. In addition, countermeasures on the system level can be taken to limit the consequences of successful SCA, rather than preventing the actual attack. We will consider examples for all three approaches in this section.

4.3.1 Amplitude-based Countermeasures

The Signal to Noise Ratio (SNR) determines the success rate of a side-channel attack—the lower the SNR, the more measurements will be needed in general.

Balanced Logic Styles A first way to reduce the SNR is to reduce the power of the leakage signal by balancing the power consumption. As a simple example, assume all results are computed on differential signals (i.e., a bit a and the complement $\overline{a}$ are always used at the same time). Then, ideally, the leakage should vanish: For example, assuming a HW model, we would always observe $\mathrm{HW}(a) + \mathrm{HW}(\overline{a}) = \text{const}$, independent of the value of a. There is a variety of such balanced logic styles[2]. However, due to various effects in real ICs (different routing delays, capacitance, process variations in general), the countermeasures can never fully remove the side-channel leakage. Yet, they help to make SCA, especially in combination with other countermeasures, much more difficult.

Another (rather obvious) example for a countermeasure based on reducing the information in the signal is protection against SPA, for example for SAM-like algorithms. Instead of using a specific implementation for squaring that produces a distinguishable pattern, multiplication can be used for both operations. However, a squaring realized with multiplication may still be recognizable, e.g., because of different memory access patterns. One solution is to use a Square-and-Multiply-always approach, where the result of the multiplication is discarded if the current exponent bit is zero. Other, more refined algorithms like the Montgomery ladder [Mon87] ensure that the same number of squares and multiplications is executed (independent of the exponent) by using two result registers. Note however that SPA-protected algorithms are usually still vulnerable to DPA.

Noise Generation Reducing the SNR can alternatively be achieved by increasing the amount of noise. A hardware or software-based noise generator can be used for this purpose, but it should be noted that averaging will allow the adversary to remove this noise again. In other words, noise generation (when used as the only countermeasure) will only increase the number of required traces. However, in combination with other countermeasures (e.g. randomized timing, see below), it can severely impede SCA, for example, because an adversary can no longer re-align traces in time based on signal patterns.

Masking Instead of introducing uncorrelated noise or reducing the leakage, *masking* fully randomizes all sensitive internal variables by internally generating a fresh, random mask for each execution of an algorithm. Then, the device only leaks the masked values, which an adversary cannot predict since the mask never "leaves" the device. An early description of such a method can e.g. be found in [CJRR99].

The general idea is to combine the sensitive value x with a random mask m, e.g., using `XOR`. Then, the device performs all computations on $x \oplus m$, and unmasks the result before outputting. For operations that are linear with respect to `XOR` (e.g., `XOR`ing a round key), applying masking is trivial, since for example $(x \oplus m) \oplus k = (x \oplus k) \oplus m$. For non-linear components, e.g., an S-Box S, masking requires to generate a new, masked S-Box S_m such that $S_m(x \oplus m) = S(x) \oplus m$.

[2] For a good overview, cf. e.g. https://www.cosic.esat.kuleuven.be/ecrypt/courses/albena11/slides/amir_moradi_secure_logic_styles.pdf

In general, the masked S-Box depends on m and has to be generated on-the-fly or precomputed for all m. However, for specific cases, e.g., the AES S-Box, algebraic properties can be utilized for more efficient realization [OMPR05, CB08]. Masking can be understood as being based on secret sharing and multiparty computation, which has lead to newer developments like so-called *threshold implementations* [NRR06].

From the attacker's point of view, masking schemes can be attacked in various ways: first of all, if the random mask m is biased, attacks may still succeed with a larger number of traces. Furthermore, *higher-order* SCA (where multiple sample points are combined to exploit combined leakage of sensitive value and mask) allows to mount DPA or CPA also for masked implementations [OMHT06].

4.3.2 Timing-based Countermeasures

In contrast to amplitude-based approaches, timing-based countermeasures lower the SNR by spreading the leakage over multiple clock cycles. Again, there are various ways to achieve this goal: first, the clock signal can be made unstable on purpose, or even randomly frequency-modulated so that a targeted operation is executed at different points in time for subsequent executions of the algorithm. It should be noted that an adversary may be able to re-align a set of traces, e.g., by extracting the peaks in the traces, using pattern matching, or FM-demodulating the traces.

Instead of manipulating the shape of the clock signal, dummy operations (or dummy cycles) can be added to the control flow to ensure that a target intermediate value is handled in different clock cycles for different executions. The dummy operations should obviously be indistinguishable from "real" operations, otherwise the adversary may be able to remove them, again using pattern matching techniques.

A slightly more algorithm-specific approach with similar ideas is to randomly change the order of operations ("shuffling"), e.g., the S-Box instances of the AES. Of course, this is only possible for operations that have no dependency (e.g., S-Boxes applied byte-wise) or that are commutative (e.g., when `XOR`ing or adding several values). A comprehensive discussion can for example be found in [VCMKS12].

It should be noted that timing-based countermeasures generally linearly reduce the attack efficiency, i.e., if a target sample is spread over τ samples, then the efficiency decreases by a factor of τ. However, by averaging (integrating) all potential target samples (in the simplest case summing τ samples), this factor becomes $\sqrt{\tau}$. Hence, in practice, protecting an implementation by time randomization alone would require a very large parameter τ. Therefore, they are usually used in combination with other countermeasures.

As a final note, an equally (or potentially even more) important aspect is the protection against timing attacks that exploit secret-dependent runtime variations. It is mandatory that the cryptographic algorithm itself has constant runtime for any combination of inputs (i.e., usually key and plaintext). Amongst others, bitslicing (Section 2.3) can be used to achieve this goal. The timing randomization then is applied on top of the constant-runtime algorithm, e.g., by inserting dummy cycles. The entropy for the randomization must be independent of the inputs to the algorithm—otherwise, if input data is used to derive "randomness" for countermeasures, a trivial timing vulnerability may be inserted into the algorithm.

4.3.3 System Level Countermeasures

First and foremost, a central countermeasure on the system level is to *diversify* keys, especially for devices that the adversary may get physical access to. This implies that every device gets a unique key—if an adversary manages to extract this key, only one specific device is affected. What happens when this principle is not followed can be see by the example of a digital locking system [SDK+13, OSS+13]. In this case, a symmetric master key was present in every locking cylinder of a complete installation (a very large building complex). By attacking a single lock, the adversary can create a device to unlock any door and access any room in the entire installation.

This is a typical example of a Single Point of Failure (SPOF), where a single successful attack has far-reaching consequences. Incidentally, note that attacks on the hardware level are only one of many ways such a master key could be leaked. Other possibilities include insiders (e.g., displeased or bribed employees), network-level attacks and rootkits (key is stored on a vulnerable computer), or unfortunate accidents (backup coy of key is stored on a decommissioned harddrive).

In contrary, for systems with proper key diversification, this problem can be largely avoided, although in general there will likely remain some valuable secrets, yet these are handled in the backend and can be protected by a variety of countermeasures. For example, in the case of a public transport card, proper key diversification ensured that a successful SCA did not scale to the compromise of the entire system, but only affected the targeted card [Osw13].

Note that in other cases, e.g., the protection or encryption of device firmware and bitstreams [MOPS13, SBO+15, SMOP14, SW12, MKP12], key diversification is not always applicable. Even if keys are diversified, an adversary who solely wants to extract Intellectual Property (IP) (e.g., to clone a device) succeeds if he can break the protection on one single device. In these case, strong classical countermeasures are hence of particular importance.

There is a variety of other possible approaches, some implemented on the actual device, others in the backend. To name a few examples: The backend system of a locking system could check for inconsistent access attempts (e.g., two distant doors were accessed approximately at the same time by the same user) to detect cloned tokens. The same applies to payment systems, where shadow accounts can be used to verify that the balance spent with a particular payment card agrees with the amount charged by the user. In general, devices which are suspected to have been compromised can be blocked. In addition, implementation attacks can be detected (or mitigated) on the side of the device, e.g., by blocking suspicious activities like the acquisition of many side-channel traces (attempt or rate limiting).

In general, system-level countermeasures strongly depend on the particular application and should hence be seen as a “second line of defense”, should the traditional countermeasures be overcome.

Bibliography

[AARR03] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM Side-Channel(s). In *Cryptographic Hardware and Embedded Systems – CHES'02*, LNCS, pages 29–45. Springer, 2003.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES'04*, volume 3156 of *LNCS*, pages 16–29. Springer, 2004.

[BDL97] Dan Boneh, Richard A. Demillo, and Richard J. Lipton. On the Importance of Checking Computations. In *Proceedings of Eurocrypt'97*, pages 37 – 51, 1997.

[Ber] Daniel J Bernstein. Cache-timing attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[Ber06] Daniel J Bernstein. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography – PKC 2006*, pages 207–228. Springer, 2006.

[Bih97] Eli Biham. A Fast New DES Implementation in Software. In *Proceedings of the 4th International Workshop on Fast Software Encryption*, FSE'97, pages 260–272, London, UK, 1997. Springer.

[BKL+07] A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES'07*, volume 4727 of *LNCS*, pages 450–466. Springer, 2007.

[Bod] Maarten Bodewes. Stackoverflow: Simulate int variables with byte or short. https://stackoverflow.com/questions/13947638/simulate-int-variables-with-byte-or-short.

[BS97] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology – CRYPTO'97*, pages 513 – 525, 1997.

[CB08] David Canright and Lejla Batina. A very compact "perfectly masked" S-box for AES. In *Applied Cryptography and Network Security*, pages 446–459. Springer, 2008.

[CFA+12] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography, Second Edition.* Chapman & Hall/CRC, 2nd edition, 2012.

[CJRR99] Suresh Chari, Charanjit S Jutla, Josyula R Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Advances in Cryptology – CRYPTO'99*, pages 398–412. Springer, 1999.

[DR99] Joan Daemen and Vincent Rijmen. AES Proposal: Rijndael, 1999. http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf.

[FH08] J. Ferrigno and M. Hlavac. When AES blinks: introducing optical side channel. *Information Security, IET*, 2(3):94–98, 2008.

[HCN$^+$06] Hagai Bar-El Hamid, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. In *Proceedings of the IEEE*, volume 94, 2006.

[HMH$^+$14] Gesine Hinterwälder, Amir Moradi, Michael Hutter, Peter Schwabe, and Christof Paar. Full-size high-security ecc implementation on msp430 microcontrollers. In *Progress in Cryptology – LATINCRYPT 2014*, pages 31–47. Springer, 2014.

[Joy09] Marc Joye. Protecting RSA against fault attacks: The embedding method. In *Fault Diagnosis and Tolerance in Cryptography FDTC'2009*, pages 41–45. IEEE, 2009.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Advances in Cryptology – CRYPTO'99*, LNCS, pages 388–397. Springer, 1999.

[KO63] A. Karatsuba and Y. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. *Nauk SSSR*, pages 293–294, 1963.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[MKP12] Amir Moradi, Markus Kasper, and Christof Paar. Black-Box Side-Channel Attacks Highlight the Importance of Countermeasures - An Analysis of the Xilinx Virtex-4 and Virtex-5 Bitstream Encryption Mechanism. In *CT-RSA'12*, volume 7178 of *LNCS*, pages 1–18. Springer, 2012.

[Mon87] Peter L Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.

[MOPS13] Amir Moradi, David Oswald, Christof Paar, and Pawel Swierczynski. Side-channel attacks on the bitstream encryption mechanism of Altera Stratix II: facilitating black-box analysis using software reverse-engineering. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays – FPGA'13*, pages 91–100, New York, NY, USA, 2013. ACM.

[MvOV96] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[NIS] NIST. FIPS 46-3 Data Encryption Standard (DES). http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *Information and Communications Security*, pages 529–545. Springer, 2006.

[OMHT06] Elisabeth Oswald, Stefan Mangard, Christoph Herbst, and Stefan Tillich. Practical second-order DPA attacks for masked smart card implementations of block ciphers. In *Topics in Cryptology – CT-RSA'06*, pages 192–207. Springer, 2006.

[OMPR05] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A side-channel analysis resistant description of the AES S-box. In *Fast Software Encryption – FSE'05*, pages 413–423. Springer, 2005.

[OSS$^+$13] David Oswald, Daehyun Strobel, Falk Schellenberg, Timo Kasper, and Christof Paar. When Reverse-Engineering Meets Side-Channel Analysis – Digital Lockpicking in Practice. In *Selected Areas in Cryptography – SAC'13*, LNCS. Springer, 2013.

[Osw13] David Oswald. *Implementation Attacks: From Theory to Practice*. PhD thesis, Ruhr-University Bochum, September 2013.

[Pro13] Emmanuel Prouf. Side Channel Attacks against Block Ciphers Implementations and Countermeasures. Tutorial at CHES'13, 2013. http://www.chesworkshop.org/ches2013/presentations/ProuffTutorialCHES2013.pdf.

[SBO$^+$15] Daehyun Strobel, Florian Bache, David Oswald, Falk Schellenberg, and Christof Paar. SCANDALee: A Side-ChANnel-based DisAssembLer using Local Electromagnetic Emanations. In *Design, Automation and Test in Europe – DATE'15*, 2015.

[SDK$^+$13] Daehyun Strobel, Benedikt Driessen, Timo Kasper, Gregor Leander, David Oswald, Falk Schellenberg, and Christof Paar. Fuming Acid and Cryptanalysis: Handy Tools for Overcoming a Digital Locking and Access Control System. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO'13*, volume 8042 of *LNCS*, pages 147–164. Springer, 2013.

[SMOP14] Pawel Swierczynski, Amir Moradi, David Oswald, and Christof Paar. Physical Security Evaluation of the Bitstream Encryption Mechanism of Altera Stratix II and Stratix III FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 7(4):34:1–34:23, December 2014.

[SNK$^+$12] Alexander Schlösser, Dmitry Nedospasov, Juliane Krämer, Susanna Orlic, and Jean-Pierre Seifert. Simple Photonic Emission Analysis of AES. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES'12*, volume 7428 of *LNCS*, pages 41–57. Springer, 2012.

[SS17] Peter Schwabe and Ko Stoffelen. All the AES You Need on Cortex-M3 and M4. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptography – SAC 2016*, pages 180–194, Cham, 2017. Springer.

[ST04] Adi Shamir and Eran Tromer. Acoustic cryptanalysis – On nosy people and noisy machines. Website, 2004. http://tau.ac.il/~tromer/acoustic/.

[STA$^+$10] Jörn-Marc Schmidt, Michael Tunstall, Roberto Maria Avanzi, Ilya Kizhvatov, Timo Kasper, and David Oswald. Combined Implementation Attack Resistant Exponentiation. In *LATINCRYPT'10*, volume 6212 of *LNCS*, pages 305–322. Springer, 2010.

[SW12] Sergei Skorobogatov and Christopher Woods. Breakthrough silicon scanning discovers backdoor in military chip. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES'12*, volume 7428 of *LNCS*, pages 23–40. Springer, 2012.

[Tex16] Texas Instruments. *MSP430FR59xx Mixed-Signal Microcontrollers*, 2016. http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf.

[VCMKS12] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. Shuffling against side-channel attacks: A comprehensive study with cautionary note. In *Advances in Cryptology – ASIACRYPT'12*, pages 740–757. Springer, 2012.

[Wik15a] Wikipedia. Algebraic normal form — Wikipedia, The Free Encyclopedia, 2015. [Online; accessed 17-January-2016].

[Wik15b] Wikipedia. Conjunctive normal form — Wikipedia, The Free Encyclopedia, 2015. [Online; accessed 17-January-2016].

[Wik16a] Wikipedia. Disjunctive normal form — Wikipedia, The Free Encyclopedia, 2016. [Online; accessed 17-January-2016].

[Wik16b] Wikipedia. Karnaugh map — Wikipedia, The Free Encyclopedia, 2016. [Online; accessed 17-January-2016].

[Wik16c] Wikipedia. Pearson product-moment correlation coefficient — Wikipedia, The Free Encyclopedia, 2016. Online; accessed 14-March-2016, https://en.wikipedia.org/w/index.php?title=Pearson_product-moment_correlation_coefficient&oldid=702729237.

List of Abbreviations

3DES Triple DES

AC Alternating Current

AES Advanced Encryption Standard

ANF Algebraic Normal Form

CNF Conjunctive Normal Form

CPA Correlation Power Analysis

CPU Central Processing Unit

CRT Chinese Remainder Theorem

DES Data Encryption Standard

DC Direct Current

DNF Disjunctive Normal Form

DPA Differential Power Analysis

DSO Digital Storage Oscilloscope

DUT Device Under Test

ECC Elliptic Curve Cryptography

ECU Electronic Control Unit

EDE Encrypt-Decrypt-Encrypt (mode of operation)

EEPROM Electrically Erasable Programmable Read-Only Memory

EM Electro-Magnetic

FFT Fast Fourier Transform

FI Fault Injection

FIB Focused Ion Beam

FM Frequency Modulation

FRAM Ferroelectric RAM

HD Hamming Distance

HW Hamming Weight

IC Integrated Circuit

IoT Internet of Things

IP Intellectual Property

LSB Least Significant Bit

LSByte Least Significant Byte

LUT Look-Up Table

MAC Message Authentication Code

MSB Most Significant Bit

μC Microcontroller

OS Operating System

PC Personal Computer

RAM Random Access Memory

RISC Reduced Instruction Set Computer

RNG Random Number Generator

RSA Rivest Shamir and Adleman (cryptosystem)

SAM Square-and-Multiply

S-Box Substitution Box

SCA Side-Channel Analysis

SNR Signal to Noise Ratio

SPA Simple Power Analysis

SPN Substitution-Permutation Network

SPOF Single Point of Failure

UV-C Ultraviolet-C (light)

XOR Exclusive OR

List of Figures

List of Tables