Cryptographic Engineering (NWI-IMC038)

Course Notes

Jos Wetzels

a.l.g.m.wetzels@student.utwente.nl

1 Basics

Benchmarking pitfalls:

- Interrupts because your program is not running exclusively on the CPU
    - Requires median of many measurements

- Difference between *reference* cycles and actual CPU speed
    - Switch off frequency scaling and TurboBoost/TurboCore

- Hyperthreading may run another process on the same physical core
    - Switch off hyperthreading

- Getting reproducible, publicly verifiable benchmarks is hard
    - Use a public benchmarking framework (eg. SUPERCOP)

Optimization: Cryptographic optimization is not the same as generic optimization in that it may not compromise security goals by having optimization measures lead to information leakage through side-channels.

Timing leakage*:*

- **Secret-data dependent branching**: Consider the following secret data dependent conditional branching code:

```
if s then
   r ← A
else
   r ← B
end if
```

  Depending on the value of *s* the assignment of *r* can take different amounts of time (depending on the nature of computations *A* and *B*) which leaks information about *s* to an outside observer. Even if *A* and *B* themselves take the same amount of time this is not *constant time* code due to optimization possibilities such as branch prediction and instruction-caching. Hence one should never use secret data dependent branch conditions.

  A fixed version of the above example would be:

```
r ← sA + (1-s)B
```

Where *s* can be expanded to all-one and all-zero masks, XOR can be used instead of addition and AND instead of multiplication.

- **Cached Memory Access**: Memory access is routed through a cache which is small and fast memory for storing frequently accessed data. Data is stored in the cache upon load from memory and remains there until it is replaced by other data which makes loading data fast if the target data is in the cache (*cache hit*) but slow if it is not (*cache miss*). Consider the following lookup table of 32-bit integers:

Line #	**Cache Data**
0	T[0],..,T[15]
1	T[16],..,T[31]
2	(..)
15	T[240],..,T[255]

The cache consists of 16 lines of 16 32-bit integers (ie. 64 bytes) each. Consider an attacker's program running on the same CPU as the targeted crypto (though hopefully under different privilege levels assigned by the OS). The attacker's program replaces some cache lines with its own data (by performing memory loads). As the crypto continues it loads from the table again after which the attacker program loads its data again. If data retrieval is fast we have a cache hit indicating the crypto did not just load from this cache line (since the attacker data was still on the cache line) while slow data retrieval indicates a cache miss and that the crypto loaded data from this line. In this fashion an attacker can determine what lookup table entries (indexed by secret data) were accessed by the crypto and hence (partially) recover secret data. This is only the most basic form of cache-timing attacks but the general idea is to <u>not access memory at secret data dependent addresses</u>.

A fixed version of the above example would be:

```
for i from 0 to n-1 do
  d ← T[i]
  if p=i then
    r ← d
  end if
end for
```

Where data at (secret) position *p* is loaded from table T by loading all items and using arithmetic to pick the right one. Two possible problems that arise are the fact that *if* statements are not constant time and *comparisons* are not constant time so this ought to be replaced by a constant-time equivalence operator.

Note that sometimes CPU instructions can take variable amounts of time depending on their arguments and hence care should be taken to either avoid those instructions or ensure that arguments to them don't leak useful timing information.

Multicore computation: Many crypto operations are fast and trivially parallel on multiple cores so the problems often introduced by multicore computing don't apply equally to cryptographic engineering.

Vector Computations: Vector instructions perform the same operation on independent data streams, known as Single Instruction Multiple Data (SIMD), are available on most common processors, can operate on vectors of multiple data types (bytes, integer, floats, etc.) and are almost as fast as scalar instructions but process far more data. In addition data-dependent branching and variably indexed lookups (as major sources of timing attacks) are expensive in SIMD so there is a performance-based need to eliminate those which synchronizes with the security-based need to do the same.

Vectorization Problems:

- **Carry Handling**: When adding two n-bit integers the result may have (n+1) bits (n bits plus carry). Scalar additions keep the carry in a special carry flag register to use for subsequent instruction (*add-with-carry*). Vector addition, however, has to either a) use special 'carry-generating' instructions or b) lose the carry. This is not a problem in general but has implications for big-integer arithmetic.

- **Removing instruction-level parallelism**: Not vectorising means we perform multiple independent instructions and trading *data-level parallelism* (DLP) for *instruction-level parallelism* (ILP). In pipelined and multiscalar execution we need ILP while vectorization removes this.

In general we wish to parallelize on as high a level as possible.

Arithmetic in Binary Fields: Think of an n-bit register as a vector register with n 1-bit entries where the bitwise operations XOR, AND, OR, etc. are now effectively vector operations (this is called *bitslicing*). *Bitslicing* often leads to speed increases (since bitwise operations are usually very fast) but more data needs to be fitted into the same registers which leads to more loads and store (often becoming the performance bottleneck).

2 Symmetric Cryptography

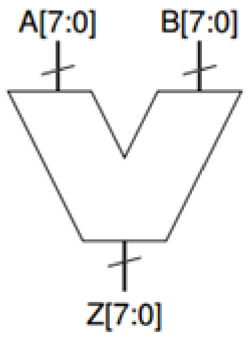
Primitives and Algorithms
Block ciphers: AES, Serpent, (3)DES, IDEA, Present, etc.
Stream ciphers: RC4, Salsa20, ChaCha20, HC-128, etc.
Hash functions: SHA-256, SHA-512, Keccak (SHA-3), Blake, etc.

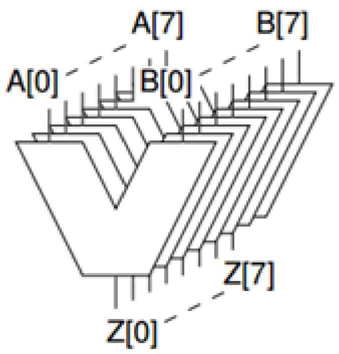
Architectures and Microarchitectures
Architectures: x86, AMD64, etc.
Microarchitectures: Pentium 4, Penryn, Sandy Bridge, etc.
Instruction-set extensions: SSE, SSE2, AVX, etc.

The Advanced Encryption Standard (AES)
Rijndael cipher proposed in 1998 selected as AES in 2000.

Block size: 128 bits (4x4 state matrix of 16 bits)
Key size: 128/192/256 bits (10/12/14 rounds)
Round key count: $n + 1$
Round operations: $SubBytes, ShiftRows, MixColumns, AddRoundKey$

Bitslicing
Lookup-table approach is inherently vulnerable to cache-timing attacks so we need to change our data representation by using *bitslicing*. Given that efficiency depends on the algorithm and microarchitecture some primitives are designed for efficient bitslicing. Essentially, bitslicing simulates a hardware implementation in software: the entire algorithm is represented as a sequence of atomic Boolean operations. Applied to AES, this means that rather than using precomputed lookup tables, the 8×8-bit S-Box as well as the linear layer are computed on-the-fly using bit-logical instructions. Since the execution time of these instructions is independent of the input values, the bitsliced implementation is inherently immune to timing attacks. Bitslicing in hardware works creating an ALU slice-by-slice instead of function by function:



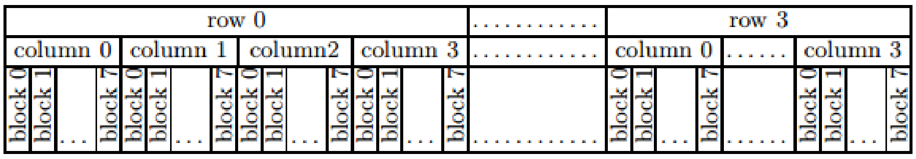


Each bit slice is a full 1-bit ALU hence requiring n slices to create an n-bit ALU.

For a software bit-slice implementation, one software logical instruction corresponds to simultaneous execution of n hardware logical gates, where n is the length of a sub-block. Hence in bitslicing we think of an n-bit register as a vector with n 1-bit entries that can be simultaneously operated upon by bitwise operations (XOR, AND, etc.).

The approach (for an AES example) is as follows:

- Process 8 AES blocks (128 bytes) in parallel
- Collect bits according to position in the byte (ie. first register contains LSBs of each byte, etc.)
- AES state stored in 8 XMM registers
- Compute 128 S-Boxes in parallel
- Never need to mix bits from different blocks



row 0																. . . .	row 3								
column 0				column 1				column2				column 3				. . . .	column 0				. . . . . .	column 3			
block 0	block 1	. . .	block 7	block 0	block 1	. . .	block 7	block 0	block 1	. . .	block 7	block 0	block 1	. . .	block 7	. . . .	block 0	block 1	. . .	block 7	. . . . . .	block 0	block 1	. . .	block 7

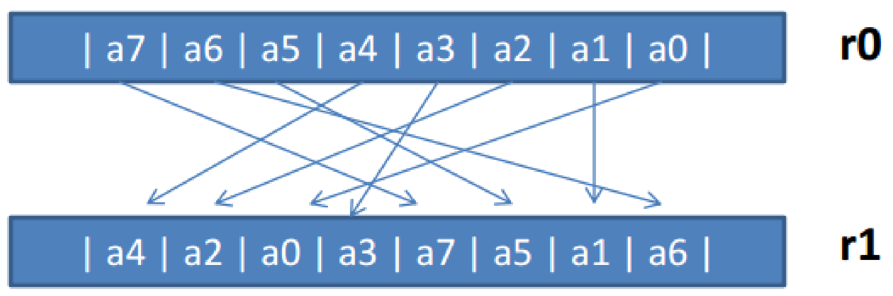
The image above depicts the bit-ordering in one 128-bit vector of the bitsliced state. Bitslicing is done by devising a sensible disposition for the state and a way to implement a bit-level vectorization of each processing step. For example, consider the following code which performs a simple left-shift on a 32-bit variable:

```
$uint\_32t\ v[];$
$v[offset_{v1}] = (v[offset_{v1}] \ll 4);$
```

We can bitslice it by translating it into 32 reassignments (from http://blog.secyoure.com/en/article/378/untwisted-bit-sliced-tea-time):

```
$shift = 4;$
for $(i = 31; i \geq 0; i$--)
 $v_{l4}[i] = (i \geq shift)?\, v[offset_{v1} + i - shift]: 0;$
```

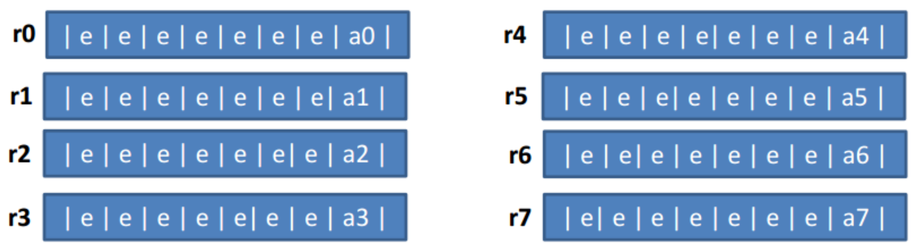
Consider the following bit-level permutation of a registry:


| a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 |
r0
| a4 | a2 | a0 | a3 | a7 | a5 | a1 | a6 |
r1

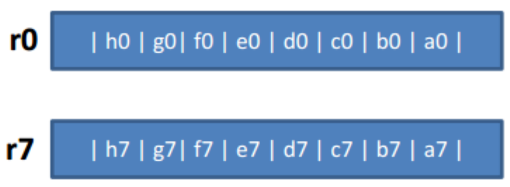
And the following AVR assembly to achieve it:

```
BST R0, 0 # store bit 0 of R0 in T flag
BLD R1, 5 # load bit 0 of R1 from T flag
..etc
```

Given that on AVR every instruction takes a clock cycle registry permutation in this fashion takes 16 clock cycles. If we use bitslicing, however, we don't presume the 8 relevant bits are stored in a single register but we store each bit individually in a different 8-bit register. So instead we do this (where *e* stands for empty):


r0 | e | e | e | e | e | e | e | a0 |
r1 | e | e | e | e | e | e | e| a1 |
r2 | e | e | e | e | e | e| e | a2 |
r3 | e | e | e | e | e| e | e | a3 |
r4 | e | e | e | e| e | e | e | a4 |
r5 | e | e | e| e | e | e | e | a5 |
r6 | e | e| e | e | e | e | e | a6 |
r7 | e| e | e | e | e | e | e | a7 |

Here registers $R_0, .., R_7$ actually represent bit positions $0, .., 7$ so now we don't have to move bits around but can 'rename' registers (eg. since bit position 0 moves to position 5 we can 'rename' R_0 to R_5). This will require 0 clock cycles since there's no such thing as 'renaming'. We simply treat R_0 as R_5 post-permutation. This allows for the processing of w (where w is the register word bit-size) blocks in parallel in the following fashion for $w = 8$:


r0 | h0 | g0| f0 | e0 | d0 | c0 | b0 | a0 |
r7 | h7 | g7| f7 | e7 | d7 | c7 | b7 | a7 |

Now ‘renaming’ R_0 to R_5 will swap 8 bits in parallel. All bitwise Boolean operations (AND, XOR, etc.) can be done w-parallel in this fashion. In this way functionality like S-Boxes can be rewritten as Boolean functions of their truth tables which saves memory (and vulnerable lookups!). The throughput of a bitsliced implementation is $T = \frac{\#bits\ processed}{\#clock\ cycles}$ eg. for 8-bit registers a function of two instructions taking a single clock cycle each has a throughput of $T = \frac{8}{2} = 4$.

Do note that bitslicing for size w requires w registers.

In summary:

- AES best case: hardware support (eg. AESNI)
- If not:
    - Bitslicing (depending on architecture)
    - Vector-permute instructions (depending on architecture and instruction set extensions)
    - Table-based approach is fast but vulnerable to timing attacks

From AES to Salsa20
Given that high-speed AES is usually for streaming modes (eg. CTR) due to the larger degree of parallelism offered by it we can also use stream ciphers (which ought to be faster). Given that the traditional stream cipher, RC4, has serious weaknesses (and can be considered practically broken for serious intents and purposes) one should look at the better suited candidates in the eSTREAM portfolio, eg. Salsa20 by djb.

Salsa20 is an ARX stream cipher working on 32-bit integers generating a random stream in 64-bit independent blocks. Salsa20 offers 4-way data-level parallelism (which translates to 4-way instruction-level parallelism in a scalar implementation) which results in scalar implementations being particularly suitable for pipelining while vector implementations aren’t unless the block independence is used to reduce instruction-level parallelism.

Key size: 256 bit
State size: 512 bit
Rounds: 20

3 Multiprecision Arithmetic

Asymmetric cryptography heavily relies on *big integer* (an integer is *big* if it’s not natively supported by the machine architecture) arithmetic (eg. RSA squaring and multiplication of 2048-bit numbers, ECs defined over large characteristic prime fields, etc.) which is also known as *multiprecision* arithmetic.

3.1 Multiprecision Multiplication

Schoolbook Multiplication
Schoolbook multiplication, also known as *long multiplication*, works by multiplying the multiplicand by each digit of the multiplier and then adding up all the properly shifted results:

$$
\begin{array}{c}
23958233 \\
5830 * \\
------ \\
0 = 23958233 * 0 + \\
71874699 = 23958233 * 30 + \\
191665864 = 23958233 * 800 + \\
119791165 = 23958233 * 5000 + \\
------ \\
139676498390
\end{array}
$$

Consider the following (more generalized) example involving Radix-m representation vectors. Given are n-bit vectors $\mathrm{A} = (\mathrm{a}_0, .., \mathrm{a}_{n-1})$ and $\mathrm{B} = (\mathrm{b}_0, .., \mathrm{b}_{n-1})$ where n = 4:

$$
\begin{array}{c}
(\mathrm{a}_0, \mathrm{a}_1, \mathrm{a}_2, \mathrm{a}_3) \\
(\mathrm{b}_0, \mathrm{b}_1, \mathrm{b}_2, \mathrm{b}_3) * \\
------ \\
\mathrm{c}_3 = \mathrm{A} * \mathrm{b}_3 \\
\dots \\
\mathrm{c}_0 = \mathrm{A} * \mathrm{b}_0
\end{array}
$$

Cost: n^2 M, $n - 1$ A
Schoolbook multiplication takes n^2 multiplications and $n - 1$ additions where n is the number of coefficients in the polynomial representation (ie. it concerns the multiplication of 2 n-byte numbers).

Schoolbook Squaring
When we multiply two n-bit integers A and B where $A = B$ off-diagonal products (ie. $A_i B_j$ where $i \neq j$) are performed twice (since the matrix is symmetric) and hence we only need to perform them once and reuse the result.

Cost: $\frac{1}{2}n(n + 1)$ M, $n - 1$ A
Schoolbook squaring takes $\frac{1}{2}n(n + 1)$ multiplications and $n - 1$ additions.

Hybrid Multiplication
The idea is to chop whole multiplication into smaller blocks and compute each of the smaller multiplications by the schoolbook method, later adding them up to the full result. Effectively the procedure consists of 2 nested loops: an inner loop performing operand scanning and an outer loop performing product scanning.

Karatsuba Multiplication
So far, multiplication of 2 n-byte numbers requires n^2 MULs. Karatsuba improved upon this as follows. Rewrite $A * B$ as $(A_0 + 2^m A_1)(B_0 + 2^m B_1)$ for half-size A_0, A_1, B_0, B_1 and compute:

$$A_0B_0 + 2^m(A_0B_1 + B_0A_1) + 2^{2m}A_1B_1 =$$
$$A_0B_0 + 2^m\big((A_0 + A_1)(B_0 + B_1) - A_0B_0 - A_1B_1\big) + 2^{2m}A_1B_1$$

The straightforward application of which is as follows:
Consider the multiplication of n-byte numbers:

$$A = (a_0, .., a_{n-1})$$
$$B = (b_0, .., b_{n-1})$$

We write $A = A_l + 2^{8k}A_h$ and $B = B_l + 2^{8k}B_h$ for k-byte integers A_l, A_h, B_l, B_h where $k = \frac{n}{2}$. We compute:

$$L = A_l * B_l = (l_0, .., l_{n-1})$$
$$H = A_h * B_h = (h_0, .., h_{n-1})$$
$$M = (A_l + A_h) * (B_l + B_h) = (m_0, .., m_{n-1})$$
$$A * B = L + 2^{8k}(M - L - H) + 2^{8n}H$$

Note that Karatsuba can be applied recursively (ie. for the internal multiplications that result in L, H, M) either to full-depth or to a certain level (after which all internal multiplications are done using schoolbook multiplication). Consider the following example of 1-level Karatsuba multiplication.

Consider the following example:

$$A = 12345, B = 6789$$
$$H = 12 * 6 = 72$$
$$L = 345 * 789 = 272205$$
$$M = (12 + 345) * (6 + 789) = 11538$$
$$A * B = L + b * (M - L - H) + b^2H =$$
$$272205 + (11538) * b + 72 * b^2 =$$
$$272205 + (11538) * 1000 + 72 * 1000^2 =$$
$$83810205$$

In the above example we split along a base of $b = 1000$ (hence the higher half words being 12 and 6). When working with vectors $b = 2^{8k}, b^2 = 2^{8n}$ for $k = \frac{n}{2}$.

Let us now consider an example involving the multiplication of two 192-bit multiprecision integers (each having 12 coefficients each) denoted $a = (a_{11}, .., a_0), b = (b_{11}, .., b_0)$, we will first split both into lower and higher 96-bit word halves (with 6 coefficients each) and calculate the intermediate *'middle'* half:

$$a_l = (a_5, .., a_0)$$
$$bl = (a_5, .., a_0)$$
$$a_h = (a_{11}, .., a_6)$$
$$b_h = (b_{11}, .., b_6)$$
$$a_m = a_l + a_h$$
$$b_m = b_l + b_h$$

Then (assuming non-recursive 1-level Karatsuba) we perform schoolbook multiplication on the halves (resulting in 3 192-bit mp integers):

$$L = a_l * b_l$$
$$H = a_h * b_h$$
$$M = a_m * b_m$$

Then we apply the Karatsuba identity to obtain $r = a * b$ as follows:

$$[(r_{23}, .., r_{18})][(r_{17}, .., r_{12})][(r_{11}, .., r_6)][(r_5, .., r_0)]$$

Here r is segmented into 4 segments of 6 coefficients which will be calculated as:

$$[H_{6..11}][H_{0..5} + (M_{6..11} - L_{6..11} - H_{6..11})][L_{6..11} + (M_{0..5} - L_{0..5} - H_{0..5})][L_{0..5}]$$

Cost: Karatsuba cost can be expressed by the recurrence $\mathrm{T(n)} = 3\mathrm{T}\left(\frac{\mathrm{n}}{2}\right) + O(n)$ where $\mathrm{O(n)}$ is the number of additions required to perform the Karatsuba identity additions and subtractions as well as additional 'bookkeeping cost' (arithmetic with addresses, etc.). In practice we can write the cost (separated in multiplications and additions) as: $3^i C_0(\frac{1}{2^i}n)$ M, $3^i C_1(\frac{1}{2^i}n) + 6$ A for recursion level i and half-size multiplication and addition cost functions C_0, C_1. 1-level Karatsuba multiplication ($i = 1$) where half-size multiplications are done using schoolbook takes $3 * (\frac{1}{2}n)^2$ multiplications and $3 * \left(\frac{1}{2}n - 1\right) + 6$ additions.

Subtractive Karatsuba

The subtractive Karatsuba approach avoids carry bits in the operands ($|A_l - A_h|, |B_l - B_h|$) which occur with regular (*additive*) Karatsuba in ($A_l + A_h, B_l + B_h$). We compute:

$$L = A_l * B_l = (l_0, .,, l_{n-1})$$
$$H = A_h * B_h = (h_0, .,, h_{n-1})$$
$$M = |(A_l - A_h)| * |(B_l - B_h)| = (m_0, .,, m_{n-1})$$

$$t = \begin{cases} 0\ if\ (M = (A_l - A_h) * (B_l - B_h)) \\ 1\ if\ otherwise \end{cases}$$
$$\widehat{M} = (-1)^t M = (A_l - A_h) * (B_l - B_h)$$
$$A * B = L + 2^{8k}\left(L + H - \widehat{M}\right) + 2^{8n} H$$

Conditional Negation

While the easy solution to conditional negation would be $if(b)\ a = -a$ this doesn't help for multiprecision arithmetic and would introduce a timing side-channel due to possible secret-data based conditional branching. So instead we opt for the constant-time solution:

$$Condition\ bit\ c = \begin{cases} 0xFF\ if\ 1 \\ 0x00\ if\ 0 \end{cases}$$

We XOR all values with c (we *don't* negate it) and subtract it from all bytes which saves us two NEG instructions and the zero register.

Refined Karatsuba

Karatsuba performs some additions twice which can be optimized as follows by using the *refined Karatsuba* identity:

$$L = A_l * B_l = (l_0, .., l_{n-1})$$
$$H = A_h * B_h = (h_0, .., h_{n-1})$$
$$M = (A_l + A_h) * (B_l + B_h) = (m_0, .., m_{n-1})$$
$$A * B = (1 - 2^{8k})(L - 2^{8k}H) + 2^{8k}M$$

Refined Karatsuba requires less additions while requiring the same amount of multiplications as regular Karatsuba.

Let us now consider the same example as with regular Karatsuba involving the multiplication of two 192-bit multiprecision integers (each having 12 coefficients each) denoted $a = (a_{11}, .., a_0), b = (b_{11}, .., b_0)$, we will first split both into lower and higher 96-bit word halves (with 6 coefficients each) and calculate the intermediate *'middle'* half:

$$a_l = (a_5, .., a_0)$$
$$bl = (a_5, .., a_0)$$
$$a_h = (a_{11}, .., a_6)$$
$$b_h = (b_{11}, .., b_6)$$
$$a_m = a_l + a_h$$
$$b_m = b_l + b_h$$

Then (assuming non-recursive 1-level Karatsuba) we perform schoolbook multiplication on the halves (resulting in 3 192-bit mp integers):

$$L = a_l * b_l$$
$$H = a_h * b_h$$
$$M = a_m * b_m$$

We then calculate the intermediate value T as follows:

$$T = (H_{0..5} - L_{6..11})$$

Which gets (superfluously) calculated twice in the original identity.

Then we apply the refined Karatsuba identity to obtain $r = a * b$ as follows:

$$[(r_{23},..,r_{18})][(r_{17},..,r_{12})][(r_{11},..,r_6)][(r_5,..,r_0)]$$

Here r is segmented into 4 segments of 6 coefficients which will be calculated as:

$$[H_{6..11}][(T_{6..11} + M_{6..11} - H_{6..11})][(M_{0..5} - L_{0..5} - T_{0..5})][L_{0..5}]$$

Cost: $3^i C_0(\frac{1}{2^i}n)$ M, $3^i C_1(\frac{1}{2^i}n) + 5$ A for recursion level i and half-size multiplication and addition cost functions C_0, C_1.

3.2 Finite Field Arithmetic

Radix-2^n Representation
Consider representing a 255-bit integer A. This can be done in Radix-2^{64} representation by using 4 64-bit integers (a_0, a_1, a_2, a_3) with:

$$A = \sum_{i=0}^{3} a_i \, 2^{64i}$$

Where arithmetic works just as before but with larger registers. While Radix-2^{64} works its efficiency is highly dependent on the efficiency of handling carries. So we get rid of carries by using Radix-2^{51} representation instead which represents *A* as 5 64-bit integers $(a_0, a_1, a_2, a_3, a_4)$ with:

$$A = \sum_{i=0}^{4} a_i \, 2^{51-i}$$

This gives us multiple ways to write the same integer, eg. $A = 2^{52}$ as:

$(2^{52}, 0,0,0,0)$
or as
$(0,2,0,0,0)$

We call representation $(a_0, a_1, a_2, a_3, a_4)$ *reduced* if $a_i \in [0,..,2^{52} - 1]$.
With many additions the coefficients may grow larger than 63 bits (which happens even faster with multiplication) so eventually we will have to carry *en bloc*:

```
signed long long carry = r.a[0] >> 51;
r.a[1] += carry;
carry <<= 51;
r.a[0] -= carry;
```

Big Integers and Polynomials
Since in the above example we are performing addition in $\mathbb{Z}_x$ we require no carries and to go to $\mathbb{Z}$ we simply evaluate at the radix (which is the same as carrying). By thinking of multiprecision integers as polynomials we can obtain some efficient results.

Bytes can be seen as 8-coefficient polynomials, ie. of the form $a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0$ for a bit-vector $(a_7, .., a_0)$ where a_7 is the msb and a_0 the lsb.
Multiplication of two such 8-coefficient polynomials a, b into r is done as follows (which is just assigning the coefficient results of regular polynomial multiplication):

$$r[0] = a[0] * b[0] = a[0] \wedge b[0]$$
$$r[1] = (a[0] * b[1]) + (a[1] * b[0]) = (a[0] \wedge b[1]) \oplus (a[1] \wedge b[0])$$
$$r[2] = (a[0] * b[2]) + (a[1] * b[1]) + (a[2] * b[0])$$
...
$$r[14] = (a[7] * b[7])$$

Where $a[i], b[i], r[i]$ denote the i^{th} coefficients of their respective polynomials. Note that in $GF(2)$ addition is $\oplus$ and multiplication $\wedge$. In a bitsliced approach the above coefficients $a[i], b[i]$ can have a size of n-bits instead of just a single bit.

Modular Reduction of Polynomials
Modular reduction of polynomials is the reduction of the polynomial coefficients $mod\ m$. Given an (irreducible) polynomial w then we reduce the polynomial r resulting from the multiplication as follows:

Consider $w = x^8 + x^4 + x^3 + x + 1$ and $r = r_{14}x^{14} + .. r_1x + r_0$ then since we have modular reduction with a polynomial of degree 8 we will reduce all coefficients in r corresponding to terms of degree 8 or higher in the following iterative manner:

$$r'[i-4] = r'[i-4] + r'[i]$$
$$r'[i-5] = r'[i-5] + r'[i]$$
$$r'[i-7] = r'[i-7] + r'[i]$$
$$r'[i-8] = r'[i-8] + r'[i]$$

For $i \in \{14, 13, .., 8\}$.
Modular Reduction of MP integers
We need arithmetic in finite fields (ie. reduction modulo a prime) as well as big integer arithmetic. Consider a prime $p = 2^{255} - 19$ (*Curve25519*) we know that $2^{255} \equiv 19\ mod\ p$ hence $2^{256} \equiv 38\ mod\ p$. We reduce the intermediate result r (coming from mp multiplication) as follows (where n is half the number of mp coefficients of r):

```
**for** $i = 0; i < n; i$++:
  r[i] += 38*r[i+n];
```

With the result residing in $(r[0], .., r[n-1])$.

Carrying after Multiplication of MP integers
For carrying integers a right shift (discarding the lowest bits) is used and for floating-point numbers we can use multiplication by the inverse of the radix (eg. for Radix-2^{22} we multiply by 2^{-22}) which requires rounding. For other cases (eg. double-precision) we simply add or subtract constant $(2^{52} + 2^{51})$ which will round the number to an integer according to the rounding mode (nearest, towards zero, etc.).

Carrying after multiplication for mp integers is done as follows:

```
for $i = 0; i < n - 1; i$++:
{
    r[i+1] += (r[i] >> 16);
    r[i] = r[i] & 0xFFFF;
}
r[0] += 38*(r[n-1]>>16);
r[n-1] = r[n-1] & 0xFFFF;
```

In the above the carry for each coefficient is expressed as (r[i] >> 16) (the right-shift by 16 discards the least significant bits). The carry is added to the next coefficient and the current coefficient is set to its least significant half (the & 0xFFFF bitmask). This is done for all but the last coefficients. The carry of the last coefficient is modularly reduced and added to the first coefficient. In some cases the coefficient r[0] may still be too large and require carrying to r[1] which is done by simply repeating part of the carrying loop again.

Montgomery Reduction

But what about other prime fields with primes that aren't so 'nice'? In that case we can use *Montgomery representation*. Consider the following problem: we need to multiply two n-limb (nb. l*imb* is the base size eg. 32 for those in Radix-2^{32}) big integers resulting in a $2n$-limb result t where we need to find $t \; mod \; p$. We do this using *Montgomery reduction* which works as follows:

- Let R be such that $gcd(R, p) = 1$ and $t < p * R$
- Represent element $a \in \mathbb{F}_p$ as $aR \; mod \; p$
- $aR * bR$ yields $t = abR^2$ ($2n$ limbs)
- Now compute Montgomery reduction $tR^{-1} \; mod \; p$
- For some R this is more efficient than division
- Typical radix-b representation: $R = b^n$

Montgomery reduction has a downside in that it takes some cost to transform to and from Montgomery representation hence making it only efficient if many operations are performed in Montgomery representation. The algorithm takes $n^2 + n$ multiplication instructions, n of which are "shortened" modulo b which is roughly the same to schoolbook multiplication. One can combine Montgomery reduction and schoolbook multiplication into *Montgomery Multiplication*.

Inversion

Given that inversion is typically much more expensive than multiplication efficient ECC arithmetic avoids frequent inversions (though typically not all inversions can be avoided). Two approaches to inversion can be taken:

Extended Euclidian Algorithm (EGCD)

Given two integers a and b the EGCD algorithm finds the *greatest common divisor (gcd)* of a and b and integers u, v such that $a * u + b * v = \gcd(a, b)$. To compute $a^{-1} \; mod \; p$ we use the *egcd* to compute $a * u + p * v = \gcd(a, p) = 1$ hence giving us $u \equiv a^{-1} \; mod \; p$. Given that the EGCD running time depends on the inputs (and hence is exposed to timing attacks) blinding should be taken into account, eg.:

- Multiply a by random r
- Invert and obtain $r^{-1}a^{-1}$
- Multiply by r to obtain a^{-1} again

Fermat's Little Theorem

Let p be prime then for any integer a it holds that $a^{p-1} \equiv 1 \; mod \; p$ which implies that $a^{p-2} \equiv a^{-1} \; mod \; p$. We can then invert by performing exponentiation with $(p - 2)$ which is fairly efficient given that the exponent is fixed (and hence known at compile-time) and quite some time can be spent on finding an efficient addition chain.

Libraries
While libraries offer a great deal of relief they don't know the particular modulus and cannot optimize for a fixed modulus. In addition they don't know the sequence of field operations you're computing and aren't always timing-attack protected. As such ECC speed records are achieved with hand-optimized assembly implementations.

4 Elliptic Curve Cryptography (ECC)

4.1 ECC and EC point multiplication

Elliptic Curve point multiplication (an integral part of ECC) consists of successively adding a point along an elliptic curve to itself presents two major problems:

1. Adding $P = (x_p, y_p)$ and $Q = (x_q, y_q)$ requires inversion in $\mathbb{F}_q$. Since inversions are expensive (especially constant-time inversions) we use projective coordinates.

2. Addition of $P = (x_p, y_p)$ and $Q = (x_q, y_q)$ needs to distinguish different cases:
    a. If $P = \mathcal{O}$ return Q
    b. Else if $Q = \mathcal{O}$ return P
    c. Else if $P = Q$ call doubling routine
    d. Else if $P = -Q$ return $\mathcal{O}$
    e. Else use addition formulas

    This is similar for doubling P:
    a. If $P = \mathcal{O}$ return P
    b. Else if $y_p = 0$ return $\mathcal{O}$
    c. Else use doubling formulas

    Constant-time implementations of this are horrible and simple implementations are likely to be insecure.

Solution: Montgomery Ladder
We use the Montgomery curve $E_M: By^2 = x^3 + Ax^2 + x$ in a x-coordinate-only differential addition chain known as the *Montgomery Ladder* (see later).

Advantages:
- Works on all inputs, no special cases
- Very regular structure, easy to protect against timing attacks
- Point compression/decompression for free
- Easy to implement, harder to screw up in hard-to-detect ways
- Simple implementations are likely to be correct and secure

Disadvantages:

- Not all curves can be converted to Montgomery shape
- Ladders on general Weierstrass curves are much less efficient
- We only get the *x* coordinate of the result, not suitable for signatures
- Can reconstruct *y*, but that involves some additional cost

4.2 ECC and Scalar Multiplication

The Elliptic Curve Discrete Logarithm Problem (ECDLP)
Consider a finite, abelian group G written additively. Compute $k * P$ for $k \in \mathbb{Z}$ and $P \in G$. This is the same as x^k for $x \in G^*$. Hence the same algorithms apply for scalar multiplication and exponentiation. The ECDLP is as follows: Given two points P and $Q \in \langle P \rangle$ find an integer k such that $kP = Q$. Where:

- P is a fixed system parameter
- k is the secret (private) key
- Q is the public key

Key generation needs to compute $Q = kP$ given k and P.

EC-Diffie-Hellman Key Exchange (ECDH)
Alice and Bob have keypairs (k_A, Q_A) and (k_B, Q_B) respectively. Alice and Bob exchange Q_A and Q_B and Alice computes the shared key as $K = k_A Q_B$ while Bob computes it as $K = k_B Q_A$. The above schemes all require the computation of kP (key generation and DH both require *one* scalar multiplication kP). There are various approaches to scalar multiplication as shown below.

Scalar Multiplication and Modular Exponentiation
The algorithms below are applicable as an approach to both scalar multiplication over ECs and modular exponentiation in multiplicative groups. Consider (finite, abelian) group G, written additively then scalar multiplication $k * P$ for $k \in \mathbb{Z}, P \in G$ is the same as modular exponentiation x^k for $x \in G'$ where G' is the multiplicative group.

Double-and-Add
Consider we wish to compute kP for the scalar $k = 105$ and the point P. We rewrite the scalar as the polynopmial $105 = 1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$ (yielding the corresponding vector $k = (1,1,0,1,0,0,1)$). We can then use the *double-and-add* algorithm below:

```
R ← P
for i ← $n - 2$ downto 0 do
  R ← 2R
  if $k[i] == 1$ then
    R ← R + P
  end if
end for
return R
```

Cost: $(n-1)$ D, $(m-1)$ A.

Given that the exponent is n-bits and has m 1 bits, then double-and-add takes $n-1$ doublings and $m-1$ additions. That is $\frac{n}{2}$ additions on average. No precomputation is required as P does not need to be known in advance. In the above example $n = 7, m = 4$ hence we have a cost of 6D, 3A.

Side-Channel Attacks
Given that the code-flow (and hence running time and power consumption) depends on the value of the (secret) scalar (ie. there is obvious secret-data dependent conditional branching) *double-and-add* is vulnerable to side-channel attacks (eg. power analysis, timing attacks, etc.).

Double-Scalar Double-and-Add
We can modify *double-and-add* to compute $k_1P_1 + k_2P_2$ as follows:

```
R ← $\mathcal{O}$
for i ← $\max(n_1, n_2) - 1$ downto 0 do
  R ← 2R
  if $k_1[i] == 1$ then
    R ← R + $P_1$
  end if
  if $k_2[i] == 1$ then
    R ← R + $P_2$
  end if
end for
return R
```

Cost: $\max(n_1, n_2)$ D, $(m_1 + m_2)$ A.

Side-Channel Attacks
Given that addition is still conditional upon secret-data (assuming the scalar(s) are secret) the same types of side-channel vulnerabilities exist in this version of the algorithm as in the original.

Double-and-Add Always
In order to try and eliminate side channels resulting from conditional addition we can try to always perform addition and simply discard the result (or perform addition with the neutral element) if so required:

```
R ← P
for i ← $n - 2$ downto 0 do
  R ← 2R
  if $k[i] == 1$ then
    R ← R + P
  else
    R ← R + $\mathcal{O}$
  end if
end for
return R
```

Cost: $(n - 1)$ D, $(n - 1)$ A.

Side-Channel Attacks
This approach, however, still does not execute in constant time.

Scalar Radix Representation
In the above examples we have written our scalar k in radix-2 but we can rewrite it in any radix we choose (which will yield shorter vector representation and hence fewer additions). Consider radix-3:

- We precompute table $T = (\mathcal{O}, P, 2P)$
- We write scalar k as $(k_{n-1}, .., k_0)_3$
- We compute the scalar multiplication as follows:

```
R ← $T[k[n - 1]]$
for i ← $n - 2$ downto 0 do
  R ← 3R
  R ← R + $T[k[i]]$
end for
return R
```

Obviously, 3 is a fairly awkward radix value so multiples of 2 would be preferable in practice.

Fixed-Window Scalar Multiplication
We can extend the above to the so-called *fixed-window scalar multiplication* method:

- Consider the fixed window width w
- We precompute table $T = (\mathcal{O}, P, 2P, .., (2^w - 1)P)$
- We write scalar k as $(k_{m-1}, .., k_0)_{2^w}$
- This is the same as chopping the binary scalar into windows of width w
- We compute the scalar multiplication as follows:

```
R ← $T[k[m-1]]$
for i ← $m-2$ downto 0 do
  for j ← 1 to $w$ do
    R ← 2R
  end for
  R ← R + $T[k[i]]$
end for
return R
```

Example
Consider the binary representation of 52348907123:

110000110000000111100111011100111 0011

and a window size of $w = 4$ then the scalar recoding would looks as follows:

Partitioning: $|1100||0011||0000||0011||1100||1110||1110||0111||0011|$
Recoding: $(12,3,0,3,12,14,14,7,3)_{2^4}$
Reconstruction = $12 * 2^{4*8} + 3 * 2^{4*7} + .. + 3 * 2^{4*0} = 52348907123$

Consider *Fixed-Window Modular Exponentiation* using square-and-multiply for $w = 4$, a 32-bit base a and a $m = 32$ coefficient (8-bit each) mp integer exponent e. We precompute $T = (a^0, .., a^{15})$ and set R ← $T[0] = a^0 = 1$. Then for every 8-bit coefficient $e[i]$, say $e[i] = 11011010_2 = 218_{10}$ we apply window partitioning: $|1101||1010|$. In order to select the first (left-to-right) window we take $e[i] \gg 4 = 1101_2 = 13_{10}$ and in order to select the second window we take $e[i] \,\&\, 0xF = e[i] \,\&\, 1111_2 = 1010_2 = 10_{10}$. This gives us $T[13] = a^{13}, T[10] = a^{10}$. Preceding each multiplication with $w = 4$ squarings gives us $R = \left(a^{0*(2^4)} * T[13]\right)^{2^4} * T[10] = a^{13*16} * a^{10} = a^{218}$.

Now let us consider the code for the same scenario but with an arbitrary 32-bit modulus m. We start by precomputing $T = (a^0, .., a^{15})$. Given that each coefficient is 8 bits our fixed window has two ‘slots’ (and hence two iterations), giving the following code:

```
R ← $a^0 = 1$
**for** i ← $m - 1$ **downto** 0 **do**
  **for** j ← 1 **to** $w$ **do**
    R ← R*R % m
  **end for**
  R ← R * $T[e[i] \gg 4]$ % m

  **for** j ← 1 **to** $w$ **do**
    R ← R*R % m
  **end for**
  R ← R * $T[e[i] \,\&\, 0xF]$ % m
**end for**
**return** R
```

In the above code we initialize the result to $T[0] = a^0 = 1$ and iterate over all coefficients of the scalar. For each coefficient we have two (since they are 8-bit each and we have $w = 4$) window computations consisting of 4 squarings followed by multiplication with the window slot over the secret scalar (first the higher then the lower 4 bits).

Cost: $(n - 1)$ D, $\left\lceil \frac{n}{w} \right\rceil$ A. In precomputation: $(\frac{1}{2}w - 1)$ D, $(\frac{1}{2}w - 1)$ A (for n-bit scalar)

This means that for larger w we require more precomputation while for smaller w we require more additions in the main algorithm. It is recommend to choose $w = 4$ or $w = 5$ for scalars ≈ 256-bit.

Side-Channel Attacks
While for each window we performing w doublings and one addition the above isn't necessarily constant time since addition (eg. for $\mathcal{O}$) might not be running in constant time and table lookups generally aren't constant time either.

Sliding-Window Scalar Multiplication
Fixed-window scalar multiplication is limited in the sense that its window size is fixed which can sometimes lead to inefficient doubling and adding sequences. In order to address this we "slide" the window over the scalar using the so-called *sliding-window scalar multiplication* approach:

- Consider window width w
- We rewrite scalar k as $(k_0, .., k_m)$ with $k_i \in \{0,1,3,5, .., 2^w - 1\}$ (ie. the odd integers) where we skip consecutive zeros while scanning from the right to the left. For example for $w = 3$ and $k = 241$ we have $k = (|1| \;\; |111| \;\; 000 \;\; |1|)_2$.

- We precompute table $T = (\mathcal{O}, P, 3P, 5P, .., (2^w - 1)P)$
- We compute the scalar multiplication as follows:

```
R ← $\mathcal{O}$
**for** i ← $m$ **downto** 0 **do**
  R ← 2R
  R ← R + $T[k[i]]$
**end for**
**return** R
```

Example
Consider the binary representation of 52348907123:

110000110000001111001110111001110011

and a window size of $w = 4$ then the scalar recoding would looks as follows:

Partitioning: $|11|0000|11|000000|1111|00|111|0|111|00|111|00|11|$
Recoding: (3,3,15,7,7,7,3)
Reconstruction = $3 * 2^{34} + 3 * 2^{28} + .. + 3 * 2^0 = 52348907123$

Consider *Sliding-Window Modular Exponentiation* using square-and-multiply for $w = 4$, a 32-bit base a and a $m = 32$ coefficient (8-bit each) mp integer exponent e. We precompute $T = (a^1, a^3, a^5, .., a^{15})$ and set R ← 1.

Then we generate the sliding window for the exponent e by iterating over all coefficients and for every coefficient storing the binary expansion in a slide array. We do this by setting $slide[i][j] = (s[i] \gg j)\&1$ which stores each of the 8 bits of $e[i]$ in a separate slide slot.

We then iterate over the entire slide and for each slot we check if $(slide[i][j] = 1)$ and if so we add all the values of $slide[i][j + 1]$ to $slide[i][j + w - 1]$ to it and set them to 0 after adding them, ie.: $slide[i][j] = sum(slide[i][j + b] \ll b), b \in (0, .., w - 1)$. To perform the square-and-multiply we simply iterate over the entire slide and perform a squaring for each iteration and if the current slide value is 1 we perform multiplication with $t[slide[i] \gg 1]$.

Eg. consider $e[i] = 11011010_2 = 218_{10}$ then we have $slide[i][7]..slide[i][0] =$ [1,1,0,1,1,0,1,0] after expansion and $slide[i][1] = 1101$ after which we have [1,1,0,0,0,0,1101,0], hence $slide[i][6] = 11$ and we have [11,0,0,0,0,0,1101,0]. Etc.

Cost: $(n - 1)$ D, $\left\lfloor \frac{n}{w+1} \right\rfloor$ A. In precomputation: $(2^{w-1} - 1)$ D, $(2^{w-1} - 1)$ A

Here we have, for the same w as with the fixed-window approach, only half the pre-computation cost and fewer additions in the main loop.

Side-Channel Attacks
This code does not, however, run in constant time and as such is vulnerable to side-channel attacks. It is, however, suitable (in *double-scalar* version) for non-secret scalar uses such as signature verification.

The Montgomery Ladder
The *Montgomery ladder* computes, if implemented correctly, scalar multiplication in a *fixed amount of time*. Consider the scalar k in the same representation as in the above examples:

```
$R_0 \leftarrow \mathcal{O}$
$R_1 \leftarrow P$
**for** i $\leftarrow n-1$ **downto** 0 **do**
  **if** $k[i] = 1$ **then**
    $R_0 \leftarrow R_0 + R_1$
    $R_1 \leftarrow 2R_1$
  **else**
    $R_1 \leftarrow R_0 + R_1$
    $R_0 \leftarrow 2R_0$
  **end if**
**end for**
**return** $R_0$
```

This can also be written as:

- Consider the n-bit scalar $k \in \mathbb{Z}$ $(k \geq 0)$ and the x-coordinate x_p of point P.

```
$X_1 \leftarrow x_p$
$X_2 \leftarrow 1$
$Z_2 \leftarrow 0$
$X_3 \leftarrow x_p$
$Z_3 \leftarrow 1$
**for** i $\leftarrow n-1$ **downto** 0 **do**
  **if** $k[i] = 1$ **then**
    $(X_3, Z_3, X_2, Z_2) \leftarrow ladderstep(X_1, X_3, Z_3, X_2, Z_2)$
  **else**
    $(X_2, Z_2, X_3, Z_3) \leftarrow ladderstep(X_1, X_2, Z_2, X_3, Z_3)$
  **end if**
**end for**
**return** $(X_2, Z_2)$
```

Advantages
Some of the advantages of the Montgomery ladder are:

- It has a very regular structure and is easy to protect against timing attacks (by replacing the if statement by a conditional swap)

- It is very fast, offers free point compression/decompression and is easy to implement.

Side-Channel Attacks
The Montgomery ladder has in effect the same speed as the double-and-add approach except that it computes the same number of point additions and doublings regardless of the value of the (secret) scalar *k* and hence does not leak timing or power information at this level. The implementation is, however, vulnerable to an (L3) *cache-based timing attack* (also known as a FLUSH+RELOAD side-channel attack).The best known example of this is the attack recovering OpenSSL ECDSA nonce by Yarom et al. The implementation of the Montgomery ladder in the relevant OpenSSL version (0.9.8) looked as follows:

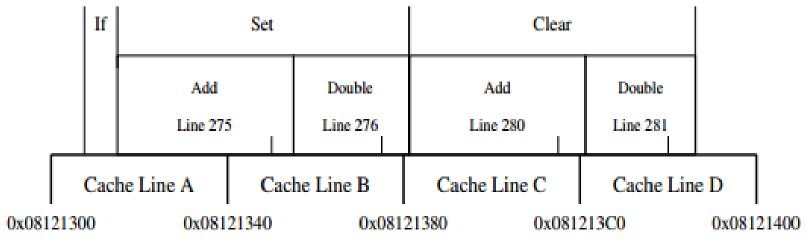
```
for (; i >= 0; i--)
{
 while (mask)
 {
  if (scalar->d[i] & mask)
  {
   //ladderstep(x1, z1, x2, z2)
   if (!gf2m_Madd(group, &point->X, x1, z1, x2, z2, ctx)) goto err;
   if (!gf2m_Mdouble(group, x2, z2, ctx)) goto err;
  }
  else
  {
   //ladderstep(x2, z2, x1, z1)
   if (!gf2m_Madd(group, &point->X, x2, z2, x1, z1, ctx)) goto err;
   if (!gf2m_Mdouble(group, x1, z1, ctx)) goto err;
  }
  mask >>= 1;
 }
 mask = BN_TBIT;
}
```

The implementation looks a little different from the above since the outer loop traverses all words (the size of which is determined by the system architecture) in the scalar while the inner loop travers all bits in each word (using a mask) but is effectively identical.

The vulnerability here lies in that while the implementation executes exactly the same sequence of operations for each bit the (only) difference between set and clear bits are the lines that invoke these operations and the targets of these operations which is sufficient for mounting an attack that recovers the values of the bits. The FLUSH+RELOAD cache-based timing attack is used to distinguish between execution of the two conditional branches (and hence reconstruct the bit of the secret scalar) as follows (as taken from the paper by Yarom et al.):

- The attack operates by dividing time into slots. At the beginning of a time slot, the spy program flushes the monitored memory line from the cache of the processor. At the end of the slot, the spy program loads data from the memory line. Loading data from cached memory lines is significantly faster than loading them from memory. Hence, by measuring the time it takes to load the data, the spy program can know whether the line is cached or not. As the line is flushed at the beginning of the slot, having it cached at the end indicates that the processor accessed the line during the time slot.

- The particular lines of code in the conditional branches map to a virtual memory address range spanning 4 cache lines A, B, C, D:



- The minimum sequence of memory line accesses required for executing this code can now be constructed. Using the FLUSH+RELOAD attack, a spy program can trace or monitor memory read and execute access of a victim program to shared memory pages. The spy program only requires read access to the shared memory pages, hence pages containing binary code in executable files and in shared libraries are susceptible to the attack.

One way to mitigate this attack is to prevent data flow from secret data to branching conditions as is done by the NaCl library. Instead of using branches, NaCl uses arithmetic operations to select the arguments and targets of the group operation. Consequently, NaCl's use of the cache is independent of the values of the bits of the secret. An example of a fixed implementation (as discussed in the TweetNacl paper) is as follows:

```
$X_1 \leftarrow x_p$
$X_2 \leftarrow 1$
$Z_2 \leftarrow 0$
$X_3 \leftarrow x_p$
$Z_3 \leftarrow 1$
**for** i $\leftarrow n - 1$ **downto** 0 **do**
  conditional_swap($(X_3, Z_3)$, $(X_2, Z_2)$, $k[i]$)
  $(X_3, Z_3, X_2, Z_2) \leftarrow ladderstep(X_1, X_3, Z_3, X_2, Z_2)$
  conditional_swap($(X_3, Z_3)$, $(X_2, Z_2)$, $k[i]$)
**end for**
**return** $(X_2, Z_2)$
```

In the above example the *conditional_swap* function performs a constant-time conditional swap of p (X_3, Z_3) and q (X_2, Z_2) depending on the scalar bit $(k[i])$. Since the entire algorithm is constant time and there is no access to different memory segments as a result from secret-data dependent conditional branching this implementation is side-channel protected.

5 Hardware Perspective

Terminology
Clock: The base time for the memories, flip flop and registers. The maximum clock frequency f_{clk} of a synchronous sequential circuit is limited by:

- Propagation delays of FFs and gates
- Setup and hold times in FFs

The minimum clock delay T_{clk} is the inverse of the maximum clock frequency $T_{clk} = \frac{1}{f_{clk}}$.

State machine: change states in rising edge clocks (they are implemented with memories).

Synchronous circuits: The entire circuit runs on the same clock, making it easy to synchronize and predict timing results.

Asynchronous circuits: Circuits are not always using the same clock which means they can achieve lower energy and better timing results but most of the time these results do not justify the associated difficulties.

5.1 Design Parameters

Hardware implementations of cryptographic components have 5 core optimization goals:

- *Area*: The implementation ‘size’ (which correlates positively with production cost).
- *Throughput*: The amount of bits the implementation can process within a given amount of time.
- *Power*: The power consumption of the implementation.
- *Energy*: The energy consumption of the implementation.
- *Security*: The security properties (eg. supported security parameters such as key lengths, protection against side-channel attacks, etc.) of the implementation.

The above optimization goals translate in corresponding *design parameters*.

Area
For hardware implementations, area is commonly measured in mm^2 (gate or transistor counts) or so-called *slice counts* for FPGAs. For software implementations it is measured in the associated memory footprint.

Throughput
Latency: the time between the start and the completion of an event (execution time) or time to encrypt/decrypt a single block of data.

Throughput (speed): the total amount of data encrypted/ decrypted in a given time e.g. [Mb/s].

$$Throughput = \frac{block\ size * \#blocks\ processed\ in\ parallel}{Latency}$$

$$Latency = \#rounds * T_{clk}$$

Eg. consider a component processing one 128-bit blocks in parallel (each processing operation taking 10 rounds) and having a minimum clock delay of $T_{clk} = 0.0025\ ns$ then we calculate throughput as:

$$Throughput = \frac{128 * 1}{10 * 0.0025 * 10^{-9}} = 5120\ Gbit/s$$

Note that throughput figures differ for different hardware architectures and the above merely illustrates the common iterative architecture.

Power and Energy Consumption
Power and energy consumption are not the same.

Power consumption: is measured in Watt as $P = I * V$ where I is the current and V the voltage. Measurement is instantaneous and usually checked for cooling or peak performance.

Energy consumption: is measured in Joule as $J = P * execution\ time$ with the battery contents being expressed in Joule, giving an idea of how many Joules are required to get a particular task done.

Security
This is covered in the next section.

Hardware vs. Software

Hardware	**Software**
Parallel execution	Sequential execution
High development cost	Low development cost
Flexibility is low	Flexibility is high

Modeling != implementation	Modeling = implementation
Timing constraints are easy, area constraints are hard.	Timing constraints are hard, area constraints are easy(ier).

6 Physical Attacks and Counter-measures

6.1 Attacks

The taxonomy of implementation attacks is as follows:

- Active vs Passive
  - *Active*: We manipulate the target and/or its environment outside of its normal behavior. The key is obtained through exploiting abnormal behavior (eg. power glitches or laser pulses).
  - *Passive*: We only observe the target. The device operates within its specification and the key is obtained through reading hidden signals.
- Invasive vs Non-invasive
  - *Invasive*: This involves depackaging the device and sometimes making contact with the chip (eg. in bus probing) but not per se (eg. in optical attacks reading out memory cells).
  - *Non-invasive*: eg. power/EM measurements

Side-channel attacks are *passive* and *non-invasive*. Side-channel leakage is based on (non-intentional) physical information. Often optimizations enable such leakages, eg.:

- Cache: fast memory access
- Fixed computation patterns
- Square vs Multiply distinguishability

6.1.1 Power Analysis Attacks

Power Analysis Attacks are a type of Side Channel Attack in which an attacker measures the power consumption of a cryptographic device during normal execution. Power analysis attacks exploit the fact that the instantaneous power consumption of a device (built in CMOS technology) depends on the data it processes and the operations it performs. The basic steps of power analysis of any type are (as derived from Kevin Meritt's presentation on DPA against AES):

- *Identify*: Determine the relationship between secret information and instantaneous power consumption. Determine the required inputs to the system, the output values to be measured and when to capture them.

- *Extract*: Develop a method for extracting the state of the relationship information. A collection of measurements (or *traces*) can be made in a non-invasive manner.

- *Evaluate*: Use the extracted information to determine all or part of the secret information.

Simple Power Analysis (SPA)
SPA has the following basic properties:

- Based on one or (the average of) a few measurements

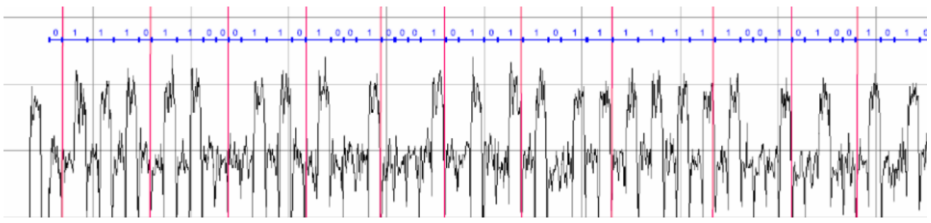
- Mostly concerns the discovery of data-(in)dependent but instruction-dependent properties eg.:
    - Symmetric: Number of rounds, memory accesses, etc.
    - Asymmetric: The key (length), etc.

- Consists of the search for repetitive patterns

Consider the following *double-and-add* ECC point multiplication code:

```
R ← P
for i ← $n - 2$ downto 0 do
  R ← 2R
  if $k[i] == 1$ then
    R ← R + P
  end if
end for
return R
```

As we can see in the line marked red, there is a conditional branch whose execution depends on the values of the secret scalar k. This means that some iterations (where $k[i] = 1$) will consist of a doubling and an addition while others only of a doubling (where $k[i] = 0$). Given that the former consumes more power than the latter an attacker analyzing a power consumption trace can determine the value of $k[i]$ at every iteration and reconstruct k.



In order to eliminate SPA vulnerability one can use the *double-and-add-always* ECC point multiplication code:

```
R ← P
**for** i ← $n - 2$ **downto** 0 **do**
  R ← 2R
  **if** $k[i] == 1$ **then**
    R ← R + P
  **else**
    R ← R + $\mathcal{O}$
  **end if**
**end for**
**return** R
```

Here addition is performed regardless of the value of $k[i]$ and hence simple power consumption patterns observed in one or a few traces (such as above) won't leak information about k. This will protect against SPA but not against other potential side-channel attacks. Avoiding conditional execution depending on secret data should be enough to mitigate SPA.

Differential Power Analysis (DPA)
DPA has the following basic properties:

- Information is recovered by inspection of the difference between traces with different (random) inputs
- Statistical methods are used to find small variations that may be overshadowed by noise or measurement errors

6.1.2 Power Analysis Counter-Measures

The purpose of counter-measures is to destroy the link between intermediate values and power consumption. This can be done by using *masking* (also known as *blinding*) and *hiding*:

- *Masking/blinding*: Masking seeks to break the relation between algorithmic and intermediate values. A random mask conceals every intermediate value (which can be done on all levels from arithmetic to gate level) by randomizing them. Since new masks are randomly chosen for each new run simple statistical analysis of power consumption does not lead to secret recovery. However 2nd and higher order DPA are still possible by looking at the joint probability distributions of multiple samples from within a trace. Some examples of masking are:

- *Boolean masking*: $x' = x \oplus r_x$
- *Arithmetic masking*: $x' = x - r_x \ mod\ 2^w$
- *Multiplicative masking*: $x' = x \otimes r_x$

Consider the following example of masked RSA (protecting against regular DPA) where message *m* is randomized:

```
$n = p * q$
$e * d = 1\ mod\ \phi(n)$
Pick random $r < n$
$m_r = m * r$
$v = m_r^e\ mod\ n$
$u = r^e\ mod\ n$
$c = v * u^{-1}\ mod\ n$
```

Or this one where private key *d* is randomized:

```
$n = p * q$
$e * d = 1\ mod\ \phi(n)$
Pick random $r < n$
$d_r = d + r * \phi(n)$
$s = m^{d_r}\ mod\ n$
```

Regarding the susceptibility to regular or higher-order DPA consider the following:

```
$f(p)$:
  $result = p \oplus secret$
  …
  $return\ c$
```

```
$f(p)$:
  $mask = rand()$
  $mp = p \oplus mask$
  $result = mp \oplus secret$
  …
  $return\ c$
```

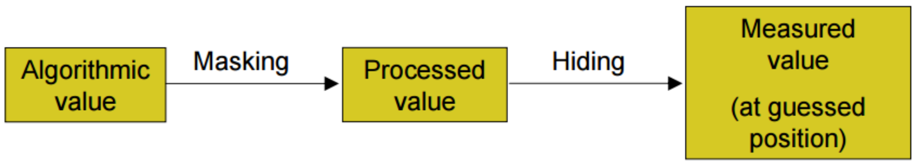
The example on the left is vulnerable to regular DPA while the example on the right isn't. The example on the right is, however, vulnerable to 2nd order DPA. Consider the points:

$$t_1 : mask = rand()$$
$$t_2 : mp = p \oplus mask$$
$$t_3 : mp \oplus secret = (p \oplus mask) \oplus secret$$

The joint distribution of the power consumption traces in points t_1, t_3 allows bit-by-bit derivation of the secret.

Usually additive arithmetic masking is the best solution as calculation of the mask correction (to *unmask* the value when needed) is always linear when using composite fields. Masking does have some issues however as it requires a TRNG and could possibly leak due to glitches if not implemented correctly.

- *Hiding*: Power consumption is made uncorrelated with intermediate values and operations which is done using eg.:
    - *Special logic styles*: duplicate logic, pre-charged circuitry, bits encoded as pairs eg. 0=(1,0) and 1=(0,1), etc. The number of bit flips is constant and data-independent.
    - *Time Randomization*: nop operations, random delays, redundant data representations, the use of dummy variables and instructions, etc.
    - *Shuffling*: Changing the order of independent instructions to reduce correlation.
    - *Power Signal Filtering*: Usage of active components (eg. transistors) in order to keep power consumption relatively constant, usage of a detached power supply, RLC filter to smooth the power consumption signal by removing high frequency components, etc.
    - *Noise generation*



DPA and ECC

Some potential counter-measures against DPA in an ECC context are (at the protocol level):

- Leakage-aware protocol design
- Scalar masking/randomization: $(k + r * l)P = k'P$
- Point masking/randomization: $kP = k(R + P) - kR$. Since P is randomized the adversary cannot predict the value of eg. $3P$

- Projective coordinates randomization

Counter-measures and Scalar Multiplication
Some potential counter-measures against SPA in a scalar multiplication context are:

- *Double-and-add-always*: Protects against SPA but not against DPA.
- *Montgomery ladder*: Protects against SPA but not against DPA.

Counter-measures and Group Operations
Some potential counter-measures against SPA in a group operation context are:

- *Side-channel atomicity*: making doubling and addition look the same, protects against SPA but not against DPA.
- *Unified addition and doubling*: One formula for both operation, protects against SPA but not against DPA.

Overview of Counter-measures

- SPA Resistance
    - Indistinguishable double and add
    - Double-and-add-always
    - Montgomery ladder
    - Window method

- DPA Resistance
    - Scalar blinding
    - Base point blinding
    - Random projective coordinates
    - Random scalar splitting
    - Random field representation