Introduction to SUNSET/FFAPL

Stefan Rass
Universität Klagenfurt
Institute of Applied Informatics
Cybersecurity

What is SUNSET/FFAPL?

Motivation of the (former) project/master thesis:

- Development of a programming language that supports operations on algebraic structures like finite fields, polynomial rings, residue classes, etc. Operations should work without explicit library calls.
- Idea: Compiler that reads protocol source code and outputs executable Java code.
- Realized: full integrated development environment with language interpreter
- Advantages:
  - Very simple handling of algebraic structures via native data types.
  - No explicit library or system calls necessary
  - Programming “close” to notation on the paper

Results

- Programming language FFAPL (Finite Field Application Language) .
- Parser and Interpreter reads, analyzes, and executes FFAPL programs.
- Integrated development environment for FFAPL code, called SUNSET.
- NSIS (Nullsoft Scriptable Install System) Windows installer for Sunset.

FFAPL – Finite Field Application Language

- Long-integer and modulo-arithmetic in (finite) algebraic structures like residue classes, polynomial rings, finite fields and elliptic curves
- Creation of (Pseudo-)random number generators
- Boolean operations: Conjunction, Disjunction, NOT.
- Comparison operators: ==, <=, >=, !=, <, >
- Control structures: While- and For-loops.
- Conditional branching: if-else-constructs
- Declaration of Functions and Procedures
- Handling sets of equivalent data types in Array.
- Handling sets of different data types in Records.
- Global constants and local variables.
- Predefined functions and procedures.
- Support of comments

Native Data Types

`String`	... Alphanumeric text
`Boolean`	... Boolean value
`Integer`	... Long integer (no numeric upper limit) hexadecimal notation `0x`... is supported
`Prime`	... Prime
`Polynomial`	... Polynomial
`Z(n)`	... Residue class modulo n
`Z(n)[x]`	... Polynomial ring modulo n
`GF(p,ply)`	... Galois Field of characteristic p and with irreducible polynomial ply
`EC(GF(…), …)`	… elliptic curve over the finite field GF and with the Weierstraß-equation determined by the coefficients a_0, a_1, a_2, a_3, a_4 und a_6 (affine coordinates required)

Special Data Types – Random Number Generators

- No notational difference to native data types
- Read-only access (like constants)
- Every access returns another random value
- Types:
  - **`PseudoRandomGenerator`**`(seed, max)`
    pseudorandom sequence, initialized with seed. Returns pseudorandom numbers between 0 (incl.) and max (incl).
  - **`RandomGenerator`**`(max)`
    Returns random numbers between 0 (incl.) and max (incl). Interface to hardware random generators prepared.
  - **`RandomGenerator`**`(min : max)`
    Returns random numbers between min (incl.) and max (incl). Interface to hardware random generators prepared.

Declaration of Constants and Variables

- Global constants
  - Read only access
  - Examples:

```
const p : Prime := 2;
const ply : Polynomial := [1 + x + x^2];
const gf : GF(p, ply) := [1 + x];
```

- Local variables:
  - Read- and Write-Access
  - Variable shadowing: local variables override global constants.
  - Examples:

```
a,b : Z(3)[x]; //polynomial ring modulo 3
primG : PseudoRandomGenerator(5, 100);
ply : Polynomial; // overrides global constant ply
f : Z(3)[][]; // two-dimensional array
r : Record a: Integer; b: Polynomial; EndRecord;
u : Z(3)[x];
```

Functions and Procedures

- Functions have a return-value (and type), procedures don‘t.
- Recursion and overloading of functions/procedures is legal
- Calls by reference and calls by value

Examples:

```
//function
function func(val1 : Integer) : Integer {
    ...
    return ... ;
}
//overloaded function
function func(val : Polynomial) : Z()[x] {
    ...
    return ... ;
}
//procedure
procedure proc(val : Integer; val2 : Polynomial) { ... }
```

Error messaging 1

- Multiple languages supported (Deutsch and English).
- Errors can be localized by row and column.

Example:

```
program calculate{
    r : Z(6);
    r := 4^-1;
}
```

causes the following German error:

```
FFapl Kompilierung: [calculate] Algebraic Error 106 (Zeile
3, Spalte 15)
Es existiert kein multiplikatives Inverses für 4 in Z(6)
```

causes the following English error:

```
FFapl compilation: [calculate] Algebraic Error 106 (line
3, column 15)
there exists no multiplicative inverse for 4 in Z(6)
```

Error messaging 2

Parser tells the expected syntax.

Example:

```
program calculate{
    r : Z(3)
}
```

causes the error

```
FFapl compilation : [calculate] ParseException 102 (Row 3,
Column 1)
"}" found in row 3, column 1. Expected one of:
    "[" ...
    ";" ...
    "[" ...
```

Console-I/O

- Console output via ***print*** or ***println***:
  ```
  x: Z(11); x := 7; println(x);
  ```
  creates the output:
  ```
  Z(11): 7
  ```
- Algebraic structure that stores the value is printed by default. To suppress this, just convert the value into a string via the pre-defined function ***str***(…):
  ```
  x: Z(11); x := 7; println(str(x));
  ```
  creates the output
  ```
  7
  ```
- Reading values from the console (user-input) can be done by the following routines: ***readInt*(…)**, ***readPoly*(…)**, ***readEC*(…)** and ***readStr*(…)**, each of which takes a string to prompt the user, and returns a value of the type specified by the suffix of the function's name (integer, polynomial, elliptic curve or string)

SUNSET

- Integrated development environment for FFAPL.
- Functionality covers:
  - File management and printing
  - Undo/Redo
  - Multi-Language support
  - Syntax- and Error-highlighting
  - Execution of FFAPL-code in separate threads
  - Interruption (abortion) of running executions
  - Individual console windows for each open FFAPL-program
  - Management of Code-Snippets
  - Integrated FFAPL-API for data types, predefined functions and snippets
  - Procedure templates and example code
  - Drag- und Drop (file opening and FFAPL-API)
  - Shortcut-Keys

Polynomials

- Polynomials are treated as literals, just like numbers:

```
p : Z(2)[x];
p := [1+x];
```

- The symbol „x“ marks the polynomial‘s variable, but using „x“ as a local program variable is allowed, even inside a polynomial literal. In that case, just enclose x into brackets:

Examples:

```
a,x: Integer;
x := 3;
a := 4;
```

syntax	evaluates to…
`[1 + 3x + x^2]`	polynomial $1+3x+x^2$
`[1 + (x)]`	1 + 3 = 4 (constant polynomial)
`[1+(a+1)x + x^2]`	$1+5x + x^2$
`[1+(x)x + x^2]`	$1 + 3x + x^2$
`[x^x]`	x^3 (exponents always evaluate)
`[(x)^x]`	27 (basis and exponent evaluated)

Construction of Finite Algebraic Structures

- Residue class groups $\mathbb{Z}_n$: **`Z`**`(n)`, arithmetic via `+`, `–`, `*`, `/` and `^`
- Residue class rings $\mathbb{Z}_n[X]$: **`Z`**`(n)[x]`, arithmetic via `+`, `–`, `*`, and `^`
- Finite fields: GF(p^n): **`GF`**`(p, ply)`, where `ply` is an irreducible (or primitive) polynomial of degree n over $\mathbb{Z}_p$, which is constructible via `ply :=` ***`irreduciblePolynomial`***`(n,p)` (predefined function). Arithmetic via `+`, `–`, `*`, `/` and `^`.
- Elliptic curves: E(F) over a finite field F and Weierstraß-equation $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$:
  `EC``(`**`GF`**`(…), `**`a1`**` := …, `**`a3`**` := …, …).`
  Points on the elliptic curve are (in affine coordinates):
  `P := <<x, y>>`, where `x`, `y` ∈ GF(…) and the Weierstraß-equation must be satisfied (type-checking done at compile- and runtime)
  Arithmetic via `+` und `*`, point at infinity symbol: `<<`**`PAI`**`>>`

Working with Variables

- In an assignment **`LHS := RHS`**, the expression **`RHS`** is always evaluated in the home algebraic structure (type) of **`LHS`**.
- Examples:
  - `m: Z(n)` causes all subsequent operations in instructions of the form `m := …` to be executed in $\mathbb{Z}_n$
  - inline expressions (e.g., in `print[ln]`-statements) can be used only if the data type can be inferred from left to right. For example:
    - `println(P + <<2,3>>)` works, since the elliptic curve on which P lives determines how to compute the + within the `println`.
    - `println(<<2,3>> + P)`, or `println(<<2,3>> + <<4,5>>)` do not work, since the host structure of `<<2,3>>` cannot be determined uniquely (the statement must be converted to the previous form)
  - Elliptic curve points admit a special syntax that allows for easy extraction of the curve coordinates; for a point P = (x, y), we can write

```
x, y: BaseGF(P); // copy the EC's base field
                 // use BaseZ for curves over Z(p)
<<x,y>> := P;  // copy the coordinates of P into x and y
```

Special Data Types 1

- Strings: explicit conversion to string via `str(…)`. Manipulation only by concatenation via +
  Example: `println("Ciphertext = " + str(c))`
- Random number generators:
  - Integer values: only declaration required, every access yields a new value:
    ```
    X : RandomGenerator(0: (2^128-1));
    for i = 1 to 10 {      // get 10 random numbers
     println("yet another AES-Key is " + X)
    }
    ```
  - On elliptic curves over the finite field $P \in GF(p^n)$ with $n \geq 1$, a special syntax is available:
    - `P := <<RandomPoint>>` delivers a random point on the curve
    - `P := <<RandomPointSubfield>>` returns a random point in the elliptic curve defined by the same equation as with (the declaration of) P, but over the subfield $\mathbb{Z}_n \subseteq GF(p^n)$

Special Data Types 2

- Arrays: 0-based; size can be set at runtime, initialization via the **new** operator:

```
arr: Z(3)[];  // array of elements from Z(3)
arr := new Z()[10];  // allocate space for 10 elements
```

  Direct declaration with values is possible:

```
a: Prime[][];     // Matrix of primes
a:= {{2,5,7},{3,11,13}};
```

- Records: unify variables of different data types

```
Certificate: Record
                 e, n: Integer; // RSA public key
                 ID: String;   // Identity
                 s: Integer;   // Signature of the CA
             EndRecord
```

Subroutine Parameter

- Finite fields are determined by several parameters (characteristic, dimension, …).
- Passing such elements to functions works by generic data types having no explicit parameters:

```
function f(m: GF; a: Z(); p: Z()[x]; C: EC) : Integer {
  // local re-construction of the finite field
  M: GF(getCharacteristic(m),getIrreduciblePolynomial(m));
  E: SameAs(C) //Type-Cloning: declare E to be the same
               //elliptic curve as C (SameAs works for all
  …            //primitive data types (not arrays or records)
}
```

- Arrays can be passes to a subroutine as well:

```
procedure p(x: Integer[]; matrix: Integer[][]) {
  n := #x; // number of elements in "x"
  n2 := #x[0];    // number of columns in "matrix"
}
```

- Records cannot be passed to subroutines as parameters!

Language Conventions

- Strict separation of declarative and procedural part
- No „early-exit“ from functions; **`return`**-Statement must always be the last instruction
- No implicit typecasting, except in these cases (only):
  - Conversion from residue class type **`Z`**`(n)` to **`Integer`** during exponentiation.
  - Conversion to string for console output.

Example: RSA-Cipher ($n = pq$, $\varphi = (p-1)(q-1)$, $m \in \mathbb{Z}_{n}$, $e,d \in \mathbb{Z}_{\varphi(n)}$)

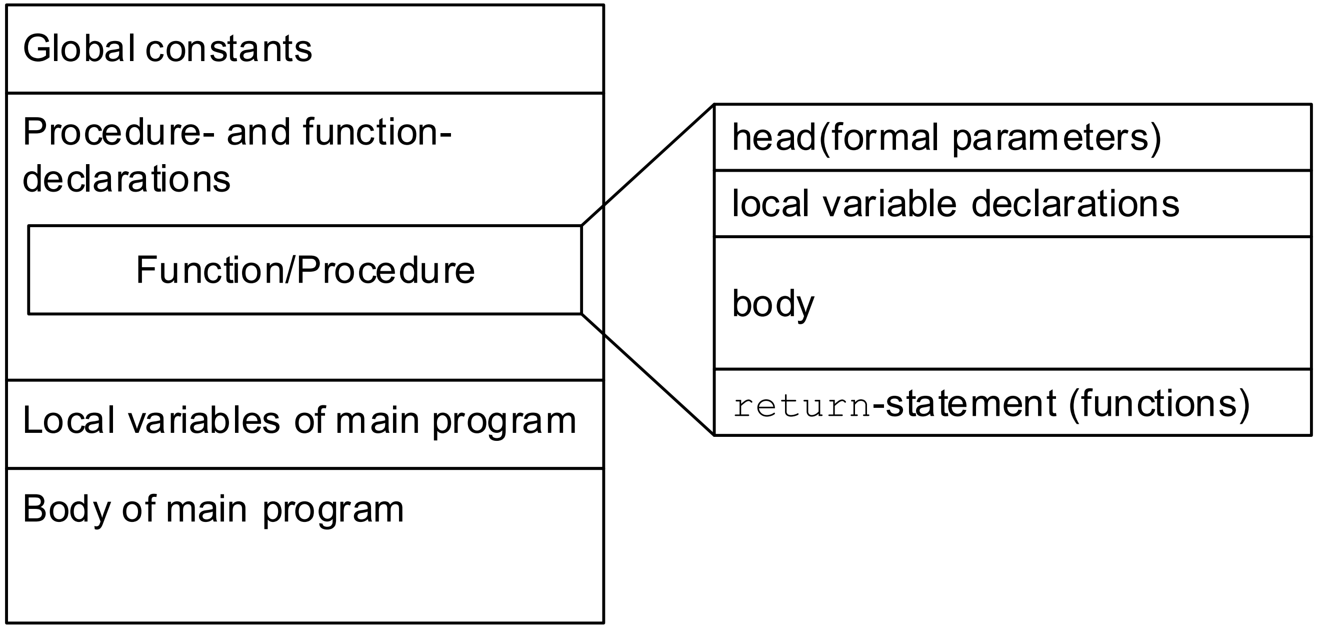
```
m, c: Z(n); e: Z(phi);
```

Upon evaluation of `c := m^e`, e is cast from **`Z`**`(phi)` to **`Integer`** (a compiler warning is issued, though).

- Explicit type-casting possible via predefined functions ***`int`***`(…)` ***`ply`***`(…)` and ***`str`***`(…)`.

Structure of FFAPL-Programs

Any FFAPL code must obey the following schema:


Global constants
Procedure- and function-declarations
Function/Procedure
Local variables of main program
Body of main program
head(formal parameters)
local variable declarations
body
return-statement (functions)

Elliptic Curve Pairings

Pairings are naturally supported by SUNSET/FFAPL via the function ***TLPairing(A, B [, n])***, but subject to the following constraints:

- **A** must be declared of type **EC(Z(p), …)**
- **B** must be declared of type **EC(GF(p, …), …)** using the same coefficients for the Weierstraß-polynomial
- In example programs, the following random choices are admissible:
  - **A := <<RandomPointSubfield>>**
  - **B := <<RandomPoint>>**
- The order of the point **A** in its home EC-group is determined brute-force, so it is advisable to pass this value as a third parameter **n** to ***TLPairing***. The curve should thus be con-structed so that the order of ist subgroups is known a-priori.

Operator Precedence

Arithmetic operations are executed as usual in mathematics, i.e., in the following sequence:

1. unitary operations (Boolean negation „`!`“, or sign change)
2. powers and exponentiations
3. `*`, `/` and **`MOD`** (note that modulo arithmetic is done implicitly by declaring the variables/expressions in the proper structure)
4. `+`, `-`
5. **`AND`**
6. **`OR`**
7. **`XOR`**
8. conditional statements (`==`, `<=`, `>=`, `!=`)

Other precedences must be enforced by embracing expressions in brackets.

Practical Part

Programming Exercises

Stefan Rass
Universität Klagenfurt
Informatik – Systemsicherheit

Chinese Remainder Theorem[1]

Theorem 2.1:

Let $m_1, m_2, \ldots, m_k \in \mathbb{N}+1$ with $(m_i,m_j) = 1$ for $i \neq j$ and $a_1, a_2, \ldots, a_k \in \mathbb{Z}$, $k \in \mathbb{N}+1$.
Then there is exactly one $x \in [0:m-1]$ satisfying
$(*)\ \ x = a_i \pmod{m_i}$, $i \in [1:k]$, $m = m_1 \cdot m_2 \cdot \ldots \cdot m_k$.

Proof:

Existence: For $n_i := m/m_i$ we have $(n_i,m_i) = 1$, so there is some x_i, for which $x_i \cdot n_i = 1 \pmod{m_i}$. With $r_i := x_i \cdot n_i$ we get for all $i \in [1:k]$ that $r_i = 0 \pmod{m_j}$ $(i \neq j)$ and $r_i = 1 \pmod{m_i}$.

$$x := \left(\sum_{i=1}^{k} a_i \cdot r_i\right) \mathrm{MOD}\ m \quad \Rightarrow \quad x = a_j \pmod{m_j} \text{ für alle } j \in [1:k];$$

so x is a solution to $(*)$.

[1] from VO „Basismechanismen der Kryptologie“, WS 2011

Chinese Remainder Theorem[1]

- Example:

  Let $m_1 = 17$, $m_2 = 21$ and $m_3 = 97$, giving the module
  $m = m_1 \cdot m_2 \cdot m_3 = 34.629$.

  With $n_i = m/m_i$ we get $n_1 = 2.037$, $n_2 = 1.649$ and $n_3 = 357$.

  By the extended Euclidian algorithm,
  $x_1 = -6$, $x_2 = 2$, $x_3 = 25$
  $r_1 = -12.222$, $r_2 = 3.298$, $r_3 = 8.925$

  If a_1, a_2 and a_3 are given, the solution is
  $x = (-12.222 \cdot a_1 + 3.298 \cdot a_2 + 8.925 \cdot a_3)$ MOD 34.629

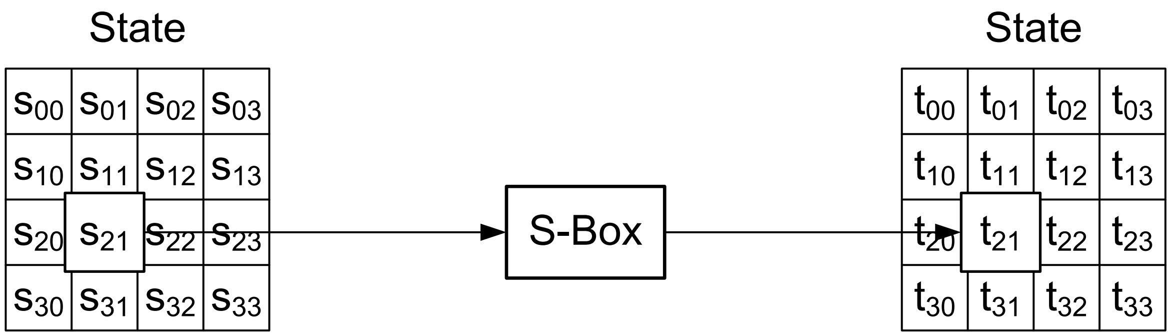
  Using $a_1 = 7$, $a_2 = 6$ and $a_3 = 25$, we get $x = 18.843$.
  From $a_1 = 2$, $a_2 = 3$ and $a_3 = 5$, we get $x = 30.075$.

[1] from VO „Basismechanismen der Kryptologie“, WS 2011

AES – SubBytes – S-Box[1]

- SubBytes is a nonlinear Byte-substition that operates on a single byte of an AES state.
- The S-Box is defined over $GF(2^8)$ with module $m(x) = x^8 + x^4 + x^3 + x + 1$, where $m(x)$ is irreducible; the S-Box can be constructed as follows:
  1. Compute the multiplicative inverse $a(x)$ of s_{ij} in $GF(2^8)$, where $\{00\} = 00_h$ is self-inverse by convention.
  2. The i-th coefficient in the result term $b(x) = t_{ij}$ is found from $a(x)$ and $c(x) = x^6 + x^5 + x + 1$ (byte-representation: {63}) as:

$$b_i = a_i \oplus a_{(i+4) \text{ MOD } 8} \oplus a_{(i+5) \text{ MOD } 8} \oplus a_{(i+6) \text{ MOD } 8} \oplus a_{(i+7) \text{ MOD } 8} \oplus c_i$$


State
s_{00} s_{01} s_{02} s_{03}
s_{10} s_{11} s_{12} s_{13}
s_{20} s_{21} s_{22} s_{23}
s_{30} s_{31} s_{32} s_{33}
S-Box
State
t_{00} t_{01} t_{02} t_{03}
t_{10} t_{11} t_{12} t_{13}
t_{20} t_{21} t_{22} t_{23}
t_{30} t_{31} t_{32} t_{33}

[1] from VO „Basismechanismen der Kryptologie“, WS 2011

AES Sbox in FFapl

```
function Sbox(s: GF) : GF {
        a, c, result: GF(2, [x^8+x^4+x^3+x+1]);
        b: Integer[];
        b := new Integer[8];
        if (s == [0]) { a := [0]; } // 0 is self-inverse
        else { a := s^-1; }
        c := [x^6 + x^5 + x + 1];
        for i = 0 to 7 {
                b[i] := (coefficientAt(a, i)
                              + coefficientAt(a, (i+4) MOD 2)
                              + coefficientAt(a, (i+5) MOD 2)
                              + coefficientAt(a, (i+6) MOD 2)
                              + coefficientAt(a, (i+7) MOD 2)
                              + coefficientAt(c, i)) MOD 2;
        }
        result := [(b[0]) + (b[1])x + (b[2])x^2 + (b[3])x^3 +
                (b[4])x^4 + (b[5])x^5 + (b[6])x^6 + (b[7])x^7];
        return result;
}
```

Guillou-Quisquater Protocol[2] 1

- Purpose: Entity A interactively proves ist identity to another entity B, by showing knowledge of a secret s_A.
- Protocol runs in rounds, each of which has 3 phases.
- Interactive zero-knowledge proof
- In the following, we consider the literal description of the protocol as found in the crypto textbook [2], and its transcription to SUNSET/FFAPL.

[2] A. Menezes, P. van Oorschot, S. Vanstone: *Handbook of Applied Cryptology*, CRC Press, 1997, p. 412

Guillou-Quisquater Protocol[2] 2

1. *Selection of system parameters*.
   (a) An authority T, trusted by all parties with respect to binding identities to public keys, selects secret RSA-like primes p and q yielding a modulus $n = pq$. (as for RSA, it must be computationally infeasible to factor n.)
   (b) T defines a public exponent $v \geq 3$ with $\gcd(v, \phi) = 1$ where $\phi = (p-1)(q-1)$ and computes its private exponent $s = v^{-1} \bmod \phi$. [...]
   (c) System parameters (v, n) are made publicly available (with guaranteed authenticity) for all users.

```
const p: Prime := getNextPrime(2^512);
const q: Prime := getNextPrime(2^100 * p);
const n: Integer := p*q;
const phi: Integer := (p-1)*(q-1);
v, s: Z(phi);
e: Integer; // auxiliary variable for constructing v
X: RandomGenerator(4:phi-1); // assure v >= 3
e := X;       // draw a random value
while(gcd(e,phi) > 1) { e := e / gcd(e,phi); }
v := e;
s := v^(-1);
```

[2] A. Menezes, P. van Oorschot, S. Vanstone: *Handbook of Applied Cryptology*, CRC Press, 1997, p. 412

Guillou-Quisquater Protocol[2] 3

2. *Selection of per-user parameters.*
   (a) Each entity A is given a unique identity I_A, from which (the *redundant identity*) $J_A = f(I_A)$, satisfying $1 < I_A < n$, is derived using a known redundancy function f [...]
   (b) T gives A the secret (*accreditation data*) $s_A = (J_A)^{-s} \bmod n$.

```
IA, JA: Integer;
sA: Z(n);
JA := f(IA); // function f assumed available
sA := JA^(-s);
```

[2] A. Menezes, P. van Oorschot, S. Vanstone: *Handbook of Applied Cryptology*, CRC Press, 1997, p. 412

Guillou-Quisquater Protocol[2] 4

3. *Protocol messages*. Each of t rounds has three messages as follows (often $t = 1$).

   $A \rightarrow B$: I_A, $x = r^v \bmod n$. (1)

   $A \leftarrow B$: e (where $1 \leq e \leq v$); (2)

   $A \rightarrow B$: $y = r \cdot s_A{}^e \bmod n$ (3)

4. *Protocol actions*. A proves its identity to B by t executions of the following; B accepts the identity only if all t executions are successful.
   (a) A selects a random secret integer r (the *commitment*), $1 \leq r \leq n - 1$, and computes (the *witness*) $x = r^v \bmod n$.
   (b) A sends to B the pair of integers (I_A, x).
   (c) B selects and sends to A a random integer e (the *challenge*), $1 \leq e \leq v$.
   (d) A computes and sends to B (the response) $y = r \cdot s_A{}^e \bmod n$.
   (e) B receives y, constructs J_A from I_A using f (see above), computes $z = J_A{}^e \cdot y^v \bmod n$, and accepts A's proof of identity if both $z = x$ and $z \neq 0$. (The latter precludes an adversary succeeding by choosing $r = 0$).

[2] A. Menezes, P. van Oorschot, S. Vanstone: *Handbook of Applied Cryptology*, CRC Press, 1997, p. 412

Guillou-Quisquater Protocol[2] 5

```
t: Integer;
r: Integer;
x,y,z: Z(n);
success: Boolean;
XR: RandomGenerator(1:n-1);        // for message (1)
XE: RandomGenerator(1:n-2);        // for message (2)
t := 10; // run 10 rounds
success := true;    // no rounds failed so far…
for i = 1 to t {
      r := XR;      // get a random value
      x := r^v;
      /* sending (IA,x) requires no action… */
      e := 1 + XE MOD int(v);    // random challenge
      y := r * sA^e;       /* compute the response */
      z := JA^e*y^v; /* check the acceptance condition */
      if (z!=x OR z==0) { success := false; }
}
// Boolean variable "success" contains the decision
```

[2] A. Menezes, P. van Oorschot, S. Vanstone: *Handbook of Applied Cryptology*, CRC Press, 1997, p. 412

Three-Party Diffie-Hellman Protocol[3] 1

```
Input: The public parameters $(G,\oplus,H,\boxplus,P,e)$ with a bilinear map $e$.
Output: the joint key $K \in H$.
    1.  $a_A \in_R \mathbb{N}$
    2.  $P_A \leftarrow [a_A]P$
    3.  send $P_A$ to $B$, $C$
    4.  receive $P_B, P_C$ from $B$, $C$
    5.  $K \leftarrow [a_A](e(P_B,P_C))$
```

- The scalar multiplication $[a]P$ can be written plainly as `a*P`.
- The group $(G,\oplus)$ is the EC group (declared below). The group $(H,\boxplus)$ is the target group of the pairing (the result type of ***TLPairing***)

```
const P: EC(GF(127,[x^2+1]), a4 := [5])
     := << [121] , [95] >>;
const g: Integer := 64;    // order of the EC subgroup
                           // generated by P
```

[3] Cohen, H. & Frey, G. (Eds.) Handbook of elliptic and hyperelliptic curve cryptography Handbook of elliptic and hyperelliptic curve cryptography, CRC Press, 2005

Three-Party Diffie-Hellman Protocol[3] 2

```
Input: The public parameters $(G,\oplus,H,\boxplus,P,e)$ with a bilinear map $e$.
Output: the joint key $K \in H$.
    1.  $a_A \in_R \mathbb{N}$
    2.  $P_A \leftarrow [a_A]P$
    3.  send $P_A$ to B, C
    4.  receive $P_B, P_C$ from B, C
    5.  $K \leftarrow [a_A](e(P_B,P_C))$
```

- For a nontrivial pairing, we require a distortion map for this group

```
// Distorsion Map
function distorsion(e : EC) : EC {
    x,y: BaseGF(e); //coordinates lie in the EC's base field
    res: SameAs(e); //the result is on the same curve as e
    << x,y >> := e; //extract the point coordinates
    res := << -x, [x]*y >>;
    return res;
}
```

[3] Cohen, H. & Frey, G. (Eds.) Handbook of elliptic and hyperelliptic curve cryptography Handbook of elliptic and hyperelliptic curve cryptography, CRC Press, 2005

Three-Party Diffie-Hellman Protocol[3] 3

```
Input: The public parameters $(G, \oplus, H, \boxplus, P, e)$ with a bilinear map $e$.
Output: the joint key $K \in H$.
    1.  $a_A \in_R \mathbb{N}$
    2.  $P_A \leftarrow [a_A]P$
    3.  send $P_A$ to B, C
    4.  receive $P_B, P_C$ from B, C
    5.  $K \leftarrow [a_A]\left(e(P_B, P_C)\right)$
```

- Declaration of variables and the protocol (next slide)

```
rng: RandomGenerator(1:g-1);
aA,aB,aC: Integer;
Pa,Pb,Pc: SameAs(P);
Ka,Kb,Kc: BaseGF(P);
```

[3] Cohen, H. & Frey, G. (Eds.) Handbook of elliptic and hyperelliptic curve cryptography Handbook of elliptic and hyperelliptic curve cryptography, CRC Press, 2005

Three-Party Diffie-Hellman Protocol[3] 4

```
Input: The public parameters $(G, \oplus, H, \boxplus, P, e)$ with a bilinear map $e$.
Output: the joint key $K \in H$.
    1.  $a_A \in_R \mathbb{N}$
    2.  $P_A \leftarrow [a_A]P$
    3.  send $P_A$ to $B$, $C$
    4.  receive $P_B, P_C$ from $B$, $C$
    5.  $K \leftarrow [a_A]\big(e(P_B, P_C)\big)$
```

```
aA := rng; aB := rng; aC := rng; // random values
Pa := aA*P; Pb := aB*P; Pc := aC*P;     // partial keys
Ka := TLPairing(Pa,distorsion(Pb))^aC;
Kb := TLPairing(Pb,distorsion(Pc))^aA;
Kc := TLPairing(Pa,distorsion(Pc))^aB;
// output (for verification)
println(str(Ka) + " == " + str(Kb) + " == " + str(Kc));
```

[3] Cohen, H. & Frey, G. (Eds.) Handbook of elliptic and hyperelliptic curve cryptography Handbook of elliptic and hyperelliptic curve cryptography, CRC Press, 2005

Open Issues and Known Bugs

Stefan Rass
Universität Klagenfurt
Informatik – Systemsicherheit

Open Issues (for future versions)

- This is an (incomprehensive) list of features that would be nice to have in future versions.
- All of these are open for implementation in a software practical, practice semester or master thesis
(please send inquiries to stefan.rass@aau.at)

- API extensions to FFAPL, including (but not limited to):
  - built-in functions for AES encryption and decryption
  - a way to define `Record` as parameter and return type for functions
  - …whatever else you may propose as useful…
- IDE extensions to SUNSET
  - Digital signatures for code
  - customizable API restrictions
  - …whatever else you may propose as useful…

Known Bugs and Limitations (as of version 2.1)

- Definitions of arrays of type `Z()[]` have parsing issues if the number of elements is not a numeric token.
  Example:
  ```
  const n: Integer := 10;
  A: Z(13)[]; // array over a polynomial ring
  // Instantiation with new operator
  A := new Z(13)[n] // this will not parse (BUG)
  ```
  Workaround: first token must be a number; add zero: `0+…`
  ```
  A := new Z(13)[0+n] // this will work (but not „[n+0]“!)
  ```
- Elliptic curve arithmetic (especially random points) works reasonably efficient only for small parameter settings
  Workaround: …unless you are willing to wait really long for the arithmetic to complete, you should do prototyping and demonstrations with small paramters with $\leq$ 10 bits

…whatever you find buggy, please report by email to stefan.rass@aau.at $\rightarrow$ thank you!