Comparative Study of Eight Formal Specifications of the Message Authenticator Algorithm

Hubert Garavel   Lina Marsso

Inria Grenoble – LIG

Université Grenoble Alpes

http://convecs.inria.fr

Outline

- The Message Authenticator Algorithm (MAA)
- Six formal models of the MAA
- Two new formal models of the MAA
- Key modelling issues
- Code generation and validation
- Errors found in ISO standards
- Conclusion

The Message Authenticator Algorithm (MAA)

Basics of cryptography

- Message Digest
  - function: (long) message → (short) numeric value
  - ensures integrity (the message has not been modified)
  - example: MD5
- Message Authentication Code (MAC)
  - function: (long) message, (short) key → (short) value
  - the key is secret, shared by the sender and the receiver
  - ensures both authentication and integrity
  - examples: hash-based (HMAC) , universal (UMAC), block ciphers (CMAC, OMAC, PMAC), etc.

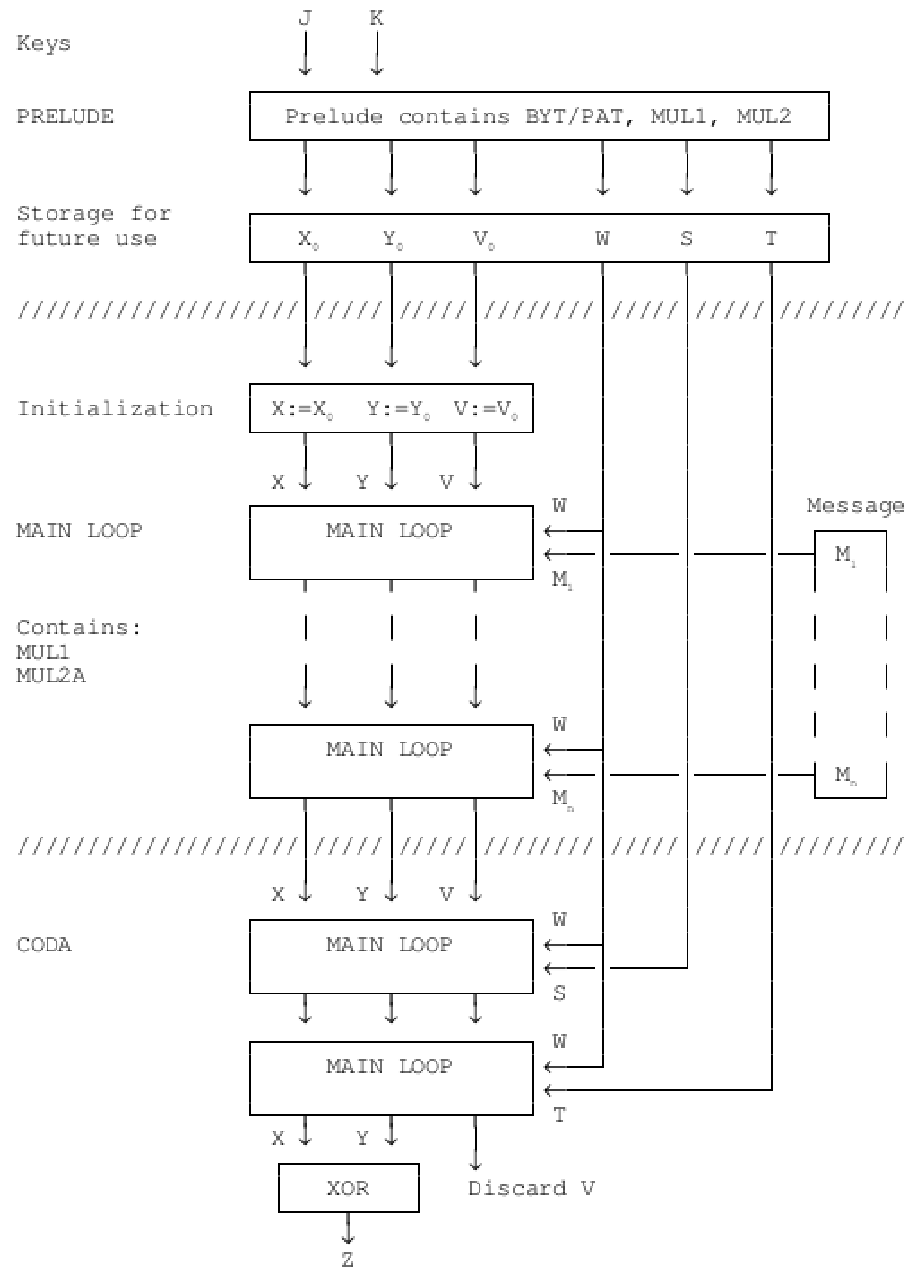
Message Authenticator Algorithm (MAA)

- First widely-used MAC function
- Designed by Donald Davies and David Clayden (NPL, 1983)
  - to protect banking transactions
  - intended to be implemented in software (32-bit PCs)
- Adopted by financial institutions
  - standardized by ISO in 1987 [ISO 8730 and 8731-2]
  - attacks published in the mid 90s
  - withdrawn from ISO standards in 2002

Overview of the MAA

- Inputs:
  - A 64-bit secret key (split into two blocks J, K)
  - A message, seen as a sequence of (less than 1,000,000) "blocks" (i.e., 32-bit words)
- Output:
  - A 32-bit MAC value (much too short nowadays!)
- Basic operations:
  - logical: AND, OR, XOR, CYC (bit rotation)
  - arithmetic: ADD, MUL (mod 2^{32}), MUL1 (mod 2^{32}-1), MUL2 (mod 2^{32}-2), MUL2A (faster variant of MUL2)

MAA data flow



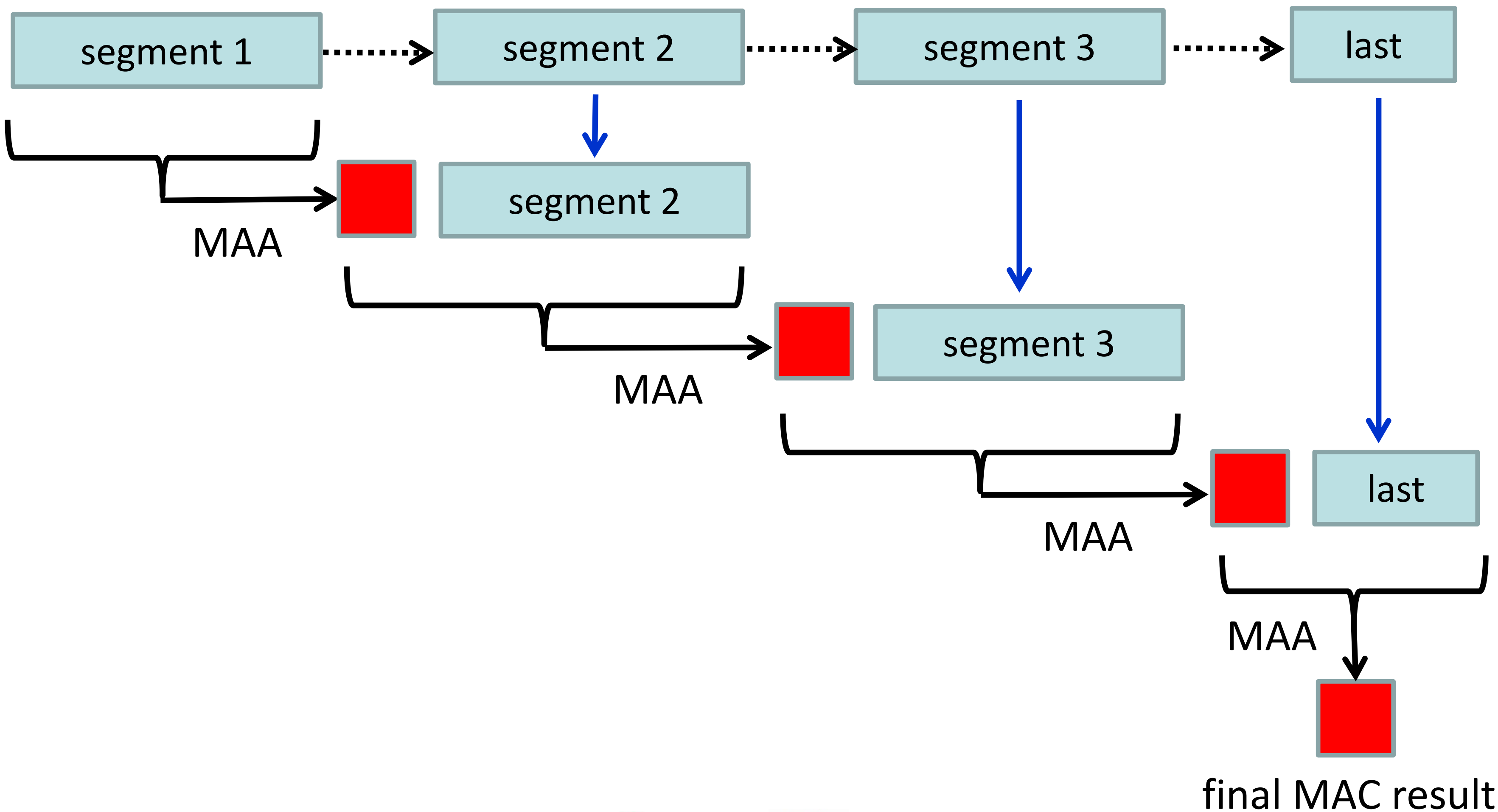
Prelude: converts key (J, K) into 6 blocks X0, Y0, V0, W, S, T

Main Loop: iterates on each message block, modifying 3 variables X, Y, V

Coda: two final iterations on the blocks S and T

"Mode of operation"

- Message is split into a list of 256-block segments



Informal specifications of the MAA

- [Davies-Clayden-88] NPL technical report
  - complete definition of the MAA in natural language
  - two implementations in C and BASIC
  - these implementations do not support the "mode of operation" (only work for messages $\leq$ 256 blocks)
- [ISO 8731-2:1992]
  - core part very similar to [Davies-Clayden-88]
- Specifications ambiguous at various places:
  - byte ordering
  - mode of operation

Test vectors for the MAA

- Various test vectors given in:
  - [Davies-Clayden-88] and [ISO 8731-2:1992]
  - [ISO-8730:1990]

J	00FF	00FF	00FF	00FF	5555	5555	5555	5555
K	0000	0000	0000	0000	5A35	D667	5A35	D667
P		FF		FF		00		00
X_0	4A64	5A01	4A64	5A01	34AC	F886	34AC	F886
Y_0	50DE	C930	50DE	C930	7397	C9AE	7397	C9AE
V_0	5CCA	3239	5CCA	3239	7201	F4DC	7201	F4DC
W	FECC	AA6E	FECC	AA6E	2829	040B	2829	040B
M_1	5555	5555	AAAA	AAAA	0000	0000	FFFF	FFFF
X	48B2	04D6	6AEB	ACF8	2FD7	6FFB	8DC8	BBDE
Y	5834	A585	9DB1	5CF6	550D	91CE	FE4E	5BDD
M_2	AAAA	AAAA	5555	5555	FFFF	FFFF	0000	0000
X	4F99	8E01	270E	EDAF	A70F	C148	CBC8	65BA
Y	BE9F	0917	B814	2629	1D10	D8D3	0297	AF6F
S	51ED	E9C7	51ED	E9C7	9E2E	7B36	9E2E	7B36
X	3449	25FC	2990	7CD8	B1CC	1CC5	3CF3	A7D2
Y	DB91	02B0	BA92	DB12	29C1	485F	160E	E9B5
T	24B6	6FB5	24B6	6FB5	1364	7149	1364	7149
X	277B	4B25	28EA	D8B3	288F	C786	D048	2465
Y	D636	250D	81D1	0CA3	9115	A558	7050	EC5E
Z	F14D	6E28	A93B	D410	B99A	62DE	A018	C83B

Why choosing the MAA?

- More challenging than conventional examples:
  - protocols and circuits deal with simple data types
  - compilers deal with abstract syntax trees (explored using standard traversals)
  - cryptographic functions exhibit "strange" behavior by performing "irregular" calculations
- Large example, still of manageable complexity
- Definition of MAA is stable and available
- MAA played a role in the history of formal methods
  - NPL developed 3 formal specifications of the MAA

Six formal models of the MAA

VDM-90 [Parkin-O'Neill] and Z-91 [Lai]

- VDM-90:
  - the first formal model of the MAA
  - included as Annex B of ISO standard 8731-2:1992
  - 3 implementations manually derived from this model: C, Miranda, Modula-2

- Z-91:
  - application of Knuth's "literate programming" idea
  - Z code fragments inserted in natural-language ISO text

LOTOS-91 [Munster]

- Only a subset of LOTOS was used:
  - abstract data types only
  - no use of the process-calculus part of LOTOS
- Equational specifications
  - sorts, operations, equations with premisses
  - fully formal
  - yet non executable
  - many "wishful-thinking" equations:

    $x = g(y) \Rightarrow f(x) = y$ means $f =_{def} g^{-1}$

A different approach

- VDM-90, Z-91, LOTOS-91 were leading edge, but:
  - "pen-and-pencil" formal methods
  - lack of validation tools
  - implementations had to be developed manually

  $\Rightarrow$ possible incompatibilities between formal models and handwritten implementations

- A different path explored at INRIA Grenoble:
  - executable formal models
  - automated translators from formal models to C

LOTOS-92 [Garavel-Turlier]

- Goals:
  - prove that LOTOS abstract data types, used under a reasonable discipline, could become executable
  - show the merits of the CAESAR.ADT compiler (LOTOS abstract data types $\rightarrow$ C)
- Features:
  - LOTOS-92: derived from LOTOS-91 with minimal changes
  - equations turned into conditional rewrite rules
  - all "wishful-thinking" equations eliminated
  - a few types and functions implemented directly in C
  - executable implementation generated by CAESAR.ADT

LNT-16 [Serwe]

- Goal:
  - effort to migrate LOTOS demo examples to LNT ones

- Features:
  - LNT-16: systematic translation of LOTOS-92 to LNT
  - slightly more concise than LOTOS-92
  - reuse of the same C code fragments as LOTOS-92
  - same test vectors, same results

LNT in a nutshell

- A safe language for message-passing concurrent systems
- A user-friendly synthesis between three paradigms:

  1) Process calculi
     - nondeterministic choice, asynchronous parallel composition, multiway rendez-vous, disruption

  2) Functional languages
     - types defined by free constructors, pattern matching

  3) Imperative languages
     - structured programming constructs (**if**, **while**, **for**, **case**, etc.), assignments, **in**/**out** parameters, Ada-like syntax for readability

- Supported by CADP: compilers, model-checkers, etc.

REC-17 [Garavel-Marsso] (1/2)

- A (conditional) term-rewrite system for the MAA
- Maybe the largest term-rewrite system available:
  - 46 pages, 1575 lines
  - 13 sorts
  - 18 constructors, 644 non-constructors
  - 684 rewrite rules
- Exhaustive, self-contained, fully formal:
  - no import of external C code
  - binary adders and multipliers for 8, 16, 32-bit words

REC-17 [Garavel-Marsso] (2/2)

- Executable:
  - automated translation to 13 languages:
    Clean, Haskell, LNT, LOTOS, Maude, mCRL2, OCaml, Opal, Rascal, Scala, Standard ML, Stratego/XT, Tom
- Verified/validated:
  - confluence
  - termination
  - all test vectors from [ISO 8731-2] and [ISO 8730]
  - new test vectors targeting endianness, byte permutations, and message segmentation

Two new formal models of the MAA

LOTOS-17 [Garavel-Marsso]

- Goals:
  - reuse the MAA knowledge acquired with REC-17
  - produce an executable LOTOS specification
  - as simple as possible
  - no need to remain aligned with LOTOS-91
- Features:
  - major rewrite, many simplifications (see the paper)
  - imports some fragments written in C (operations on 32-bit machine words)
  - (test vectors not added)

LNT-17 [Garavel-Marsso]

- Design:
  - derived from LOTOS-17
  - further simplified by using LNT's imperative style
  - extended with additional test vectors (pseudo-random message generation)
- Qualities:
  - MAA model with the most test vectors
  - very readable
  - close to the original MAA specification

Overview of MAA models

model	size (in lines)	total size
VDM-90	275	275
Z-91	608	608
LOTOS-91	438	438
LOTOS-92	641 (+ 63 lines in C)	704
LNT-16	543 (+ 63 lines in C)	606
REC-17 (+ tests)	1575	1575
LOTOS-17	266 (+ 157 lines in C)	423
LNT-17	268 (+ 345 lines in C)	345
LNT-17 (+ tests)	1334 (+ 345 lines in C)	1679

Executable specifications are not necessarily larger

Key modelling issues

Local variables in functions (1/3)

- LNT-17: imperative style, easy to write, easy to read
  - local variables and assignments
  - compute a result once and reuse it several times
  - direct correspondence with the informal MAA specification

```
function MUL1 (X, Y : Block) : Block is
   var U, L, S, C : Block in
      U := HIGH_MUL (X, Y);
      L := LOW_MUL (X, Y);
      S := ADD (U, L);
      C := CAR (U, L);
      assert (C == x00000000) or (C == x00000001);
      return ADD (S, C)
   end var
end function
```

- VDM-90: very similar style, using the "**let**" operator

Local variables in functions (2/3)

- LOTOS-91:
  - MUL1 can still be defined using a single function
  - but not executable (wishful-thinking equations)

```
opns MUL1 : Block, Block -> Block
forall X, Y, U, L, S, P: Block, C: Bit
   NatNum (X) * NatNum (Y) = NatNum (U ++ L),
   NatNum (U) + NatNum (L) = NatNum (S) + NatNum (C),
   NatNum (C + S) = NatNum (P)
   => MUL1 (X, Y) = P;
```

The 32-bit strings U and L are such that the integer value of their concatenation is equal to the 64-bit product of the integer values of the 32-bit strings X and Y.

Local variables in functions (3/3)

- LOTOS-92, REC-17:
  - this time, MUL1 is defined as an executable function
  - but it requires two auxiliary functions
  - rather far from the informal MAA specification

```
opns MUL1    : Block, Block -> Block
     MUL1_UL : Block, Block -> Block
     MUL1_SC : Block, Block -> Block
forall X, Y, U, L, S, C : Block
MUL1 (X, Y)    = MUL1_UL (HIGH_MUL (X, Y), LOW_MUL (X, Y));
MUL1_UL (U, L) = MUL1_SC (ADD (U, L), CAR (U, L));
MUL1_SC (S, C) = ADD (S, C);
```

Functions returning multiple results

- LNT-17: functions can have "**out**" or "**in out**" parameters (call by result or call by value-result)

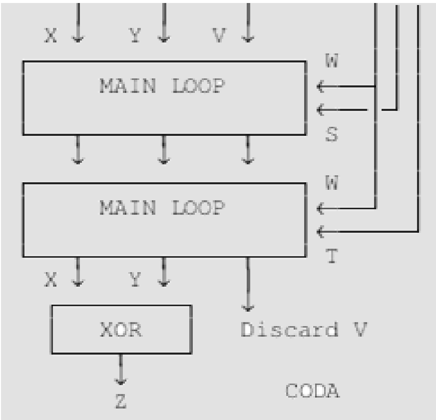
```
function Prelude (in J, K : Block, out X, Y, V, W, S, T : Block) is
    ...
end function
```

- In other languages: functions can return only one result
  - VDM-90, Z-91: Prelude returns a 6-tuple of blocks
  - LOTOS-91, LOTOS-17: Prelude returns a 3-tuple of block pairs
    $\Rightarrow$ requires auxiliary types, tupling, detupling, etc.
  - REC-17: Prelude was split into 3 functions, each returning a block pair
    $\Rightarrow$ decomposition not feasible in the general case

Useful combinations of LNT features

```
function MainLoop (in out X, Y, V : Block, W, B : Block) is
   V := CYC (V);
   var E, X1, Y1 : Block in
      E := XOR (V, W);
      X1 := MUL1 (XOR (X, B), FIX1 (ADD (XOR (Y, B), E)));
      Y1 := MUL2A (XOR (Y, B), FIX2 (ADD (XOR (X, B), E)));
      X := X1;
      Y := Y1
   end var
end function
```

```
function Coda (in var X, Y, V : Block, W, S, T : Block, out Z : Block) is
   -- Coda (two more iterations with S and T)
   MainLoop (!?X, !?Y, !?V, W, S);
   MainLoop (!?X, !?Y, !?V, W, T);
   use V;
   Z := XOR (X, Y)
end function
```



Code generation and validation

Validation

- LOTOS-17
  - compiles without warning using CAESAR.ADT and then "gcc –Wall"
  - passes tests of ISO 8730, Annexes E.3.4 and E.4
- LNT-17
  - compiles without warning using LNT2LOTOS, then CAESAR.ADT, then "gcc –Wall"
  - especially, LNT2LOTOS reports no unused variable, no useless assignment, etc.
  - passes tests of ISO 8730, Annexes E3, E.3.4, and E.4 and ISO 8731-2, Annex A

Performance improvements

- 1990: handwritten Miranda code derived from VDM-90
  - 60 seconds to process an 84-block message
  - 480 seconds to process a 588-block message

- Today: C code generated from LOTOS-17
  - 0.37 second to process a 1,000,000-block message
- Today: C code generated from LNT-17
  - 0.65 second to process a 1,000,000-block message

    (a bit slower than LOTOS since LNT-17 contains many assertions)

- "formal" and "executable" are no longer exclusive

Errors found in ISO standards

Errata: ISO-8730:1990, Annex E.2

E.2 Texte de l'exemple

Le texte du message non mis en page est

TO YOUR BANK

FROM OUR BANK

QD-80 07 14-DQ ///// 1056/ QX-127-XQ

QT-

TRNSFR USD $1234567,89 FRM ACCNT 48020-166
///// TO ACCNT 40210-178

-TQ

KEEP ON QT EXPECT VISIT ON FRIDAY OF
NEW DIV VP ON PROJECT QT-QWERT-TQ BE ~~CAREFUL~~

REGARDS BE\n\n\ \ \ Careful

QUIRTO

QK-1357BANKATOBANKB-KQ

Errata: ISO-8730:1990, Annex E.3, E.4

E.3.2 Texte à entrer dans l'algorithme

84

Ce texte est traité sous forme de ~~86~~ nombres de 4 hex (32 bits) chacun. Le caractère le plus à gauche d'un nombre est l'extrémité la plus significative, par exemple le mot BANQUE se traduit par 42 41 4E 4B en hex et 42 est l'octet le plus significatif du mot.

E.3.4 Valeurs X et Y pour un message de ~~86~~ blocs

84

84

Les valeurs X,Y pour un message de ~~86~~ blocs correspondant à D.4.2 avec la clé de l'exemple sont donnés. Pour chaque bloc du message sont indiquées les valeurs résultantes X et Y ainsi que le numéro de bloc du message (1-~~86~~). Enfin sont indiquées les étapes S et T ainsi que la valeur finale Z.

84

M = 0A 20 20 20	X = 0A D6 7E 20	Y = 30 26 14 92	N =1
M = 54 4F 20 59	X = EC E5 07 4A	Y = 30 24 B3 7F	N =2

E.4 Exemple d'un message de ~~516~~ blocs obtenu en répétant ~~six~~ fois le message de ~~86~~ blocs

588 7 84

Ce message doit être réparti en jeux de 256 blocs (1024 octets chacun) et il forme 2 jeux complets et un jeu de 4 blocs (516 = 256 + 256 + 4). Les résultats sont présentés ci-dessous, le début et la fin des deux jeux sont indiqués, le reste, une fois calculé, est simplement représenté sous forme de tirets «-- -- --». Le troisième jeu est présenté dans son entier ainsi que la valeur finale Z.

Errata: ISO-8731-2:1992, Annex A

- Incorrect test vectors given for function PAT
  [Davies-Clayden-88, Table 3] and [ISO 8732-2:1992, Table A.3]

{X0,Y0}	0103 0703 1D3B 7760	PAT{X0,Y0}	EE
{V0,W}	0103 050B 1706 5DBB	PAT{V0,W}	BB
{S,T}	0103 0705 8039 7302	PAT{S,T}	E6

should be replaced with:

{H4,H5}	0000 0003 0000 0060	PAT{H4,H5}	EE
{H6,H7}	0003 0000 0006 0000	PAT{H6,H7}	BB
{H8,H9}	0000 0005 8000 0002	PAT{H8,H9}	E6

Conclusion

Conclusion

- MAA:
  - a pioneering algorithm in cryptography (80s)
  - an early application of formal methods (90s)
  - contributions: 2 new MAA models (LOTOS-17, LNT-17)
  - a 9th MAA model in preparation: VDM-18 [Nick Battle]
- LNT:
  - the "great unification" between imperative, functional, and process-algebraic languages?
  - solves many pitfalls of traditional formal methods
  - also suitable for non-concurrent (i.e. sequential) code