Andrés Chamorro, Diego Polanco, Luís Rojas, Oscar Muñoz

TAD<List[Byte]>

List[Byte] = {Byte, List[Byte]}

Byte = $(a_0,a_1,a_2,a_3,a_4,a_5,a_6,a_7)$

siendo $a_n \in \{0, 1\}$

Invariante:

$\forall$ Byte $\in$ List[Byte], $\forall$ i $\in$ {0,1,2,3,4,5,6,7}, Byte[i] $\in$ {0,1}

Byte.length = 8

Main operations

Builder → CreateByteList(): List[Byte]

Analyzer → calculateTime(): List[Byte] → Unit

Analyzer → singleMessageDoubleChecksum(): List[Byte] x int_1 x int_2 x int_3 → Byte

Analyzer → singleMessageBlockCheckSum(): List[List[Byte]] x int_1 x int_2 x Byte_1 x Byte_2 → Byte

Analyzer → createBlocks(): List[Byte] x int → List[List[Byte]]

Analyzer → Group(): List[Byte] x int x List[List[Byte]]→ List[List[Byte]]

CreateByteList()

“Crea una lista de Bytes, la lista inicia vacía”

{pre : True}

{pos : Se retorna una List[Byte] vacía}

calculateTime(List[Byte] b)

“Mide el tiempo de cálculo del checksum para la lista de bytes b”

{pre : b ≠ nill}

{pos : Se calcula el tiempo de ejecución sin retorno}

singleMessageDoubleChecksum(List[Byte] m , Int mod, Int k, Int bs)

“Se calcula el checksum del mensaje m dividido en bloques de tamaño bs usando la operación módulo en base mod con factor de desplazamiento (‘shifting’) de k”

{pre : mod>0 ^ bs>0 ^ me ≠ nill ^ k>=0}

{pos : Se retorna el checksum de tipo List[Byte]}

singleMessageBlockCheckSum(List[List[Byte]] db, Int mod, Int k, Byte sumA, Byte sumB)

“Se calcula el checksum del mensaje ‘db’ dividido en bloques de datos usando la operación módulo en base ‘mod’ con factor de desplazamiento (‘shifting’) de k con variables para la recursión sumA y sumB”

{pre : mod>0 ^ bs>0 ^ me ≠ nill ^ k>=0 ^ sumA>0 ^ sumB>0}

{pos : Se retorna el checksum en formato List[Byte]}

createBlocks (List[Byte] l, int blocksize)

“La función createBlocks toma una lista de bytes (l) y la divide en bloques de tamaño específico por blocksize. Luego devuelve una lista de bloques de bytes que representan estos grupos”

{pre : l ≠ nill ^ blocksize >0}

{pos : retorna una lista de bloques de bytes, donde cada bloque tiene el tamaño especificado por el blocksize}

Group(List[Byte] l, Int scale, List[List[Byte] acc)

“La función group toma una lista de bytes (l), y la divide en grupos de tamaño especificado por scale. Luego, devuelve una lista de listas de bytes que representan estos grupos (acc).”

{pre : l ≠ nill ^ scale > 0}

{pos : retorna la lista de listas de bytes}

Andrés Chamorro, Diego Polanco, Luís Rojas, Oscar Muñoz

TAD<ByteString>

ByteString = {s, ByteString}

s: String ∈ {“0”,”1”}

Invariante:

∀ s ∈ ByteString, s ∈ {“0”,”1”}

ByteString.length = 8

Main operations

Builder→ normalStringToByteString(): String → ByteString

Analyzer → calculateTime(): String → Unit

Analyzer → byteStringToNormalString(): ByteString → String

Analyzer → residue(): ByteString_1 x ByteString_2 x ByteString_3 → ByteString

Analyzer → auxBinaryModulo2(): ByteString_1 x ByteString_2 → ByteString

Analyzer → Xor(): ByteString_1 x ByteString_2 → ByteString

Analyzer → verifyMessage(): ByteString_1 x ByteString_2 x ByteString_3 → ByteString

Analyzer → getMessage(): ByteString_1 x ByteString_2 x ByteString_3 → ByteString

normalStringToByteString(String n)

“ Esta función convierte una cadena de texto normal en una secuencia de bits. Primero, la cadena se convierte en una serie de bytes utilizando UTF-8. Luego, cada byte se transforma en una secuencia de bits y se concatenan para formar la ByteString resultante**”**

{pre: n ≠ nill}

{pos: retorna el ByteString}

calculateTime(String m)

“Mide el tiempo de cálculo del CRC para cadenas de texto”

{pre : m ≠ nill}

{pos : Se calcula el tiempo de ejecución sin retorno}

byteStringToNormalString(String bytes)

"Recibe una cadena de ByteString y retorna un String traducido"

{pre :la cadena bytes es un ByteString}

{pos :traduce la cadena de Bytes}

residue(String num, String den)

"calcula el residuo de una división binaria entre el 'num' (numerador) y el 'den' (denominador)"

{pre : 'num' y 'den' son ByteStrings}

{pos : retorna el residuo}

auxBinaryModulo2(String num, String den)

"es el paso recursivo para calcular el residuo de una división binaria entre el 'num' (numerador) y el 'den' (denominador)"

{pre : 'num' y 'den' son ByteStrings}

{pos : retorna el residuo}

Xor(String a, String b)

"Hace la operación xor en dos ByteString"

{pre : "a" y "b" son ByteStrings}

{pos : retorna el resultado del xor}

verifyMessage(String m, String p, String r)

"Toma un mensaje (m), un polinomio (p), y un residuo (r). Concatena el mensaje con el residuo para formar un nuevo mensaje. Luego, calcula el residuo de este nuevo mensaje utilizando el polinomio dado. Finalmente, verifica si todos los bits del residuo calculado son cero y devuelve verdadero si es así, y falso sí no**."**

{pre : 'm', 'p' y 'r' son ByteStrings}

{pos: Retorna verdadero si todos los bits del residuo calculado son cero, y falso sí no.}

getMessage(String m, String p, String r)

“Toma un mensaje (m), un polinomio (p), y un residuo (r). Concatena el mensaje con el residuo para formar un nuevo mensaje. Luego, calcula el residuo de este nuevo mensaje utilizando el polinomio dado. Finalmente, devuelve el mensaje original eliminando los bits correspondientes al residuo calculado.**”**

{pre : ‘m’, ‘p’ y ‘r’ son ByteStrings}

{pos: Retorna el mensaje original después de eliminar los bits correspondientes al residuo calculado.}