Abstract

Formal verification is getting more and more important in computer science. However the state of the art formal verification methods in cryptography are very rudimentary. These theories are one step to provide a tool box allowing the use of formal methods in every aspect of cryptography. Moreover we present a proof of concept for the feasibility of verification techniques to a standard signature algorithm.

Contents

1 Extensions to the Word theory required for SHA1 2
2 Message Padding for SHA1 4
3 Formal definition of the secure hash algorithm 4
4 Definition of rsacrypt 7
5 Lemmata for modular arithmetic 7
6 Positive differences 8
7 Lemmata for modular arithmetic with primes 9
8 Correctness proof for RSA 9
9 Extensions to the Word theory required for PSS 10
10 EMSA-PSS encoding and decoding operation 12
11 RSS-PSS encoding and decoding operation 20
1 Extensions to the Word theory required for SHA1

theory WordOperations
imports Word
begin

type-synonym bv = bit list

datatype HEX = x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF

definition bvxor: bvxor a b = bv-mapzip (op bitxor) a b

definition bvand: bvand a b = bv-mapzip (op bitand) a b

definition bvor: bvor a b = bv-mapzip (op bitor) a b

primrec last where
  last [] = Zero
| last (x#r) = (if (r=[]) then x else (last r))

primrec dellast where
  dellast [] = []
| dellast (x#r) = (if (r = []) then [] else (x#dellast r))

fun bvrol where
  bvrol a 0 = a
| bvrol [] x = []
| bvrol (x#r) (Suc n) = bvrol (r@[x]) n

fun bvrord where
  bvrord a 0 = a
| bvrord [] x = []
| bvrord x (Suc n) = bvrord (last x # dellast x) n

fun selecthelp where
  selecthelp [] n = (if (n <= 0) then [Zero] else (Zero # selecthelp [] (n-(1::nat)))))
| selecthelp (x#l) n = (if (n <= 0) then [x] else (x # selecthelp l (n-(1::nat)))))

fun select where
  select [] i n = (if (i <= 0) then (selecthelp [] n) else select [] (i-(1::nat))
(n-(1::nat))))
| select (x#l) i n = (if (i <= 0) then (selecthelp (x#l) n) else select l (i-(1::nat))
(n-(1::nat))))
definition

\texttt{addmod32}: \texttt{addmod32} \, a \, b = \\
\texttt{rev} (\texttt{select} (\texttt{rev} (\texttt{nat-to-bv} ((\texttt{bv-to-nat} a) + (\texttt{bv-to-nat} b)))) \, 0 \, 31)

definition

\texttt{bv-prepend}: \texttt{bv-prepend} \, x \, b \, \texttt{bv} = \texttt{replicate} \, x \, \texttt{b} \, \&\, \texttt{bv}

primrec \texttt{zerolist} where

zerolist 0 = \texttt{[]} \\
| \texttt{zerolist} \, (\texttt{Suc} \, n) = \texttt{zerolist} \, n \, \&\, [\texttt{Zero}]

primrec \texttt{hextofv} where

hextofv x0 = [\texttt{Zero},\texttt{Zero},\texttt{Zero},\texttt{Zero}] \\
| hextofv x1 = [\texttt{Zero},\texttt{Zero},\texttt{Zero},\texttt{One}] \\
| hextofv x2 = [\texttt{Zero},\texttt{Zero},\texttt{One},\texttt{Zero}] \\
| hextofv x3 = [\texttt{Zero},\texttt{Zero},\texttt{One},\texttt{One}] \\
| hextofv x4 = [\texttt{Zero},\texttt{One},\texttt{Zero},\texttt{Zero}] \\
| hextofv x5 = [\texttt{Zero},\texttt{One},\texttt{Zero},\texttt{One}] \\
| hextofv x6 = [\texttt{Zero},\texttt{One},\texttt{One},\texttt{Zero}] \\
| hextofv x7 = [\texttt{Zero},\texttt{One},\texttt{One},\texttt{One}] \\
| hextofv x8 = [\texttt{One},\texttt{Zero},\texttt{Zero},\texttt{Zero}] \\
| hextofv x9 = [\texttt{One},\texttt{Zero},\texttt{Zero},\texttt{One}] \\
| hextofv xA = [\texttt{One},\texttt{Zero},\texttt{One},\texttt{Zero}] \\
| hextofv xB = [\texttt{One},\texttt{Zero},\texttt{One},\texttt{One}] \\
| hextofv xC = [\texttt{One},\texttt{One},\texttt{Zero},\texttt{Zero}] \\
| hextofv xD = [\texttt{One},\texttt{One},\texttt{Zero},\texttt{One}] \\
| hextofv xE = [\texttt{One},\texttt{One},\texttt{One},\texttt{Zero}] \\
| hextofv xF = [\texttt{One},\texttt{One},\texttt{One},\texttt{One}]

primrec \texttt{hextofv} where

hextofv [\texttt{}] = [\texttt{}] \\
| hextofv (x\texttt{{--}}r) = \texttt{hextofv} \, x \, \&\, \texttt{hextofv} \, r

\textbf{lemma} \texttt{selectlenhelp}: \texttt{length} (\texttt{selecthelp} \, l \, i) = (i + 1) \\
⟨\texttt{proof}⟩

\textbf{lemma} \texttt{selectlenhelp2}: \land \, i. \, \texttt{ALL} \, l \, j. \, \texttt{EX} \, k. \, \texttt{select} \, l \, i \, j = \texttt{select} \, k \, 0 \, (j - i) \\
⟨\texttt{proof}⟩

\textbf{lemma} \texttt{selectlenhelp3}: \texttt{ALL} \, j. \, \texttt{select} \, l \, 0 \, j = \texttt{selecthelp} \, l \, j \\
⟨\texttt{proof}⟩

\textbf{lemma} \texttt{selectlen}: \texttt{length} (\texttt{select} \, l \, i \, j) = j - i + 1 \\
⟨\texttt{proof}⟩

\textbf{lemma} \texttt{addmod32len}: \land \, a \, b. \, \texttt{length} (\texttt{addmod32} \, a \, b) = 32 \\
⟨\texttt{proof}⟩

end
2 Message Padding for SHA1

theory SHA1Padding
imports WordOperations
begin

definition zero_count :: nat ⇒ nat where
  zero_count n := (((((n + 64) div 512) + 1) * 512) − n − (65::nat)

definition help_padd :: bv ⇒ bv ⇒ nat ⇒ bv where
  help_padd x y n = x @ [One] @ zero_list (zero_count n) @ zero_list (64 − length y) @ y

definition sha1_padd :: bv ⇒ bv where
  sha1_padd x = help_padd x (nat_to_bv (length x)) (length x)

end

3 Formal definition of the secure hash algorithm

theory SHA1
imports SHA1Padding
begin

We define the secure hash algorithm SHA-1 and give a proof for the length of the message digest

definition f1f where
  f1f x y z = bvor (bvand x y) (bvand (bv_not x) z)

definition fxor where
  fxor x y z = bvxor (bvxor x y) z

definition fmaj where
  fmaj x y z = bvor (bvor (bvand x y) (bvand x z)) (bvand y z)

definition fselect :: nat ⇒ bit list ⇒ bit list ⇒ bit list ⇒ bit list ⇒ bit list ⇒ bit list ⇒ bit list ⇒ bit list where
  fselect r x y z = (if (r < 20) then (f1f x y z) else (if (r < 40) then (fxor x y z) else (if (r < 60) then (fmaj x y z) else (fxor x y z))))

definition K1 where
  K1 = hex_to_bv [x5,xA,x8,x2,x7,x9,x9]

definition K2 where
  K2 = hex_to_bv [x6,xE,xD,x9,xE,xB,xA,x1]

definition K3 where
  K3 = hex_to_bv [x8,xF,x1,xB,xC,xD,xC]
definition \( K_4 \) where
\[ K_4 : K_4 = \text{hextobv} [xC, xA, x6, x2, xC, x1, xD, x6] \]

definition \( kselect :: \text{nat} \Rightarrow \text{bit list} \) where
\[ kselect : kselect r = \begin{cases} K_1 & \text{if } r < 20 \\ K_2 & \text{if } r < 40 \\ K_3 & \text{if } r < 60 \\ K_4 & \text{otherwise} \end{cases} \]

definition \( \text{getblock} \) where
\[ \text{getblock} : \text{getblock} x = \text{select} x 0 511 \]

fun \( \text{delblockhelp} \) where
\[ \text{delblockhelp} [] n = [] \\
\quad | \text{delblockhelp} (x\#r) n = (\text{if } n \leq 0 \text{ then } x\#r \text{ else delblockhelp } r (n-(1::\text{nat}))) \]

definition \( \text{delblock} \) where
\[ \text{delblock} : \text{delblock} x = \text{delblockhelp} x 512 \]

primrec \( \text{sha1compress} \) where
\[ \text{sha1compress} 0 b A B C D E = (\text{let } j = (79::\text{nat}) \text{ in} \\
\quad (\text{let } W = \text{select } b (32\ast j) ((32\ast j)+31) \text{ in} \\
\quad (\text{let } AA = \text{addmod32} (\text{addmod32} (\text{addmod32} W \\
\quad \text{berol } A 5)) (\text{fselect } j B C D)) (\text{addmod32} E (kselect j)); \\
\quad BB = A; \\
\quad CC = \text{berol } B 30; \\
\quad DD = C; \\
\quad EE = D \text{ in} \\
\quad AA\@BB@CC@DD@EE)) \\
\quad | \text{sha1compress} (\text{Suc } n) b A B C D E = (\text{let } j = (79 - (\text{Suc } n)) \text{ in} \\
\quad (\text{let } W = \text{select } b (32\ast j) ((32\ast j)+31) \text{ in} \\
\quad (\text{let } AA = \text{addmod32} (\text{addmod32} (\text{addmod32} W \\
\quad \text{berol } A 5)) (\text{fselect } j B C D)) (\text{addmod32} E (kselect j)); \\
\quad BB = A; \\
\quad CC = \text{berol } B 30; \\
\quad DD = C; \\
\quad EE = D \text{ in} \\
\quad \text{sha1compress } n b AA BB CC DD EE)) \]

definition \( \text{sha1expandhelp} \) where
\[ \text{sha1expandhelp} x i = (\text{let } j = (79+16-i) \text{ in} (\text{berol } (\text{buxor}( \\
\quad \text{select } x (32\ast(j-(3::\text{nat})))) (31+(32\ast(j-(3::\text{nat})))) (\text{select } x (32\ast(j-(8::\text{nat})))) \\
\quad (31+(32\ast(j-(8::\text{nat})))))) (\text{buxor}(\text{select } x (32\ast(j-(14::\text{nat})))) (31+(32\ast(j-(14::\text{nat})))))))) \\
\quad (\text{select } x (32\ast(j-(16::\text{nat})))) (31+(32\ast(j-(16::\text{nat})))))) 1)) \]

fun \( \text{sha1expand} \) where
\[ \text{sha1expand} x i = (\text{if } i < 16 \text{ then } x \text{ else} \\
\quad \text{let } y = \text{sha1expandhelp } x i \text{ in} \]
\[ \text{sha1expand} \ (x \oplus y) \ (i - 1) \]

definition sha1compressstart where
sha1compressstart \( r \) \( b \) \( A \) \( B \) \( C \) \( D \) \( E \) = sha1compress \( r \) (sha1expand \( b \) 79) \( A \) \( B \) \( C \) \( D \) \( E \)

function (sequential) sha1block where
sha1block \( b \) \( [] \) \( A \) \( B \) \( C \) \( D \) \( E \) = (let \( H \) = sha1compressstart 79 \( b \) \( A \) \( B \) \( C \) \( D \) \( E \);
\( AA \) = addmod32 \( A \) (select \( H \) 0 31);
\( BB \) = addmod32 \( B \) (select \( H \) 32 63);
\( CC \) = addmod32 \( C \) (select \( H \) 64 95);
\( DD \) = addmod32 \( D \) (select \( H \) 96 127);
\( EE \) = addmod32 \( E \) (select \( H \) 128 159)
in \( AA \oplus BB \oplus CC \oplus DD \oplus EE \))

sha1block \( b \) \( x \) \( A \) \( B \) \( C \) \( D \) \( E \) = (let \( H \) = sha1compressstart 79 \( b \) \( A \) \( B \) \( C \) \( D \) \( E \);
\( AA \) = addmod32 \( A \) (select \( H \) 0 31);
\( BB \) = addmod32 \( B \) (select \( H \) 32 63);
\( CC \) = addmod32 \( C \) (select \( H \) 64 95);
\( DD \) = addmod32 \( D \) (select \( H \) 96 127);
\( EE \) = addmod32 \( E \) (select \( H \) 128 159)
in sha1block (getblock \( x \)) (delblock \( x \)) \( AA \) \( BB \) \( CC \) \( DD \) \( E \))

\langle proof \rangle

termination \langle proof \rangle

definition IV1 where
IV1: \( IV1 = \text{hexvtobv} \ [x6, x7, x4, x5, x2, x3, x0, x1] \)

definition IV2 where
IV2: \( IV2 = \text{hexvtobv} \ [xE, xF, xC, xD, xA, xB, x8, x9] \)

definition IV3 where
IV3: \( IV3 = \text{hexvtobv} \ [x9, x8, xB, xA, xD, xC, xF, xE] \)

definition IV4 where
IV4: \( IV4 = \text{hexvtobv} \ [x1, x0, x3, x2, x5, x4, x7, x6] \)

definition IV5 where
IV5: \( IV5 = \text{hexvtobv} \ [xC, x3, xD, x2, xE, x1, xF, x0] \)

definition sha1 where
sha1: \( sha1 \) \( x \) = (let \( y \) = sha1padd \( x \) in
sha1block (getblock \( y \)) (delblock \( y \)) \( IV1 \) \( IV2 \) \( IV3 \) \( IV4 \) \( IV5 \))

\langle proof \rangle

lemma sha1blocklen: \( \text{length} \ (\text{sha1block} \ b \ A \ B \ C \ D \ E) = 160 \) \langle proof \rangle

lemma sha1len: \( \text{length} \ (\text{sha1} \ m) = 160 \)
4 Definition of rsa-crypt

definition rsa-crypt :: nat ⇒ nat ⇒ nat ⇒ nat
where
cryptcorrect: rsa-crypt M e n = M ^ e mod n

lemma rsa-crypt-code [code]:
rsa-crypt M e n = (if e = 0 then 1 mod n
  else if even e then rsa-crypt M (e div 2) n ^ 2 mod n
  else (M * rsa-crypt M (e div 2) n ^ 2 mod n) mod n)

end

5 Lemmata for modular arithmetic

lemma divmultassoc: a div (b*c) * (b*c) = ((a div (b * c)) * b)*c
〈proof〉

lemma delmod: (a::nat) mod (b*c) mod c = a mod c
〈proof〉

lemma timesmod1: ((x::nat)*((y::nat) mod n)) mod (n::nat) = ((x*y) mod n)
〈proof〉

lemma timesmod3: ((a mod (n::nat)) * b) mod n = (a*b) mod n
〈proof〉

lemma remaindertext: (y mod (a::nat) = z mod a) ⇒ (x*y) mod a = (x*z) mod a
〈proof〉

lemma remaindertest: ((a mod (n::nat))^i) mod n = (a ^ i) mod n
〈proof〉
6 Positive differences

theory Pdifference
imports "~/src/HOL/Number-Theory/Primes Mod"
begin

definition pdifference :: nat ⇒ nat ⇒ nat where
[simp]: pdifference a b = (if a < b then (b−a) else (a−b))

lemma timesdistributesoverpdifference:
  m∗(pdifference a b) = pdifference (m∗(a::nat)) (m∗ (b::nat))
⟨proof⟩

lemma addconst: a = (b::nat) ⇒ c+a = c+b
⟨proof⟩

lemma invers: a ≤ x ⇒ (x::nat) = x − a + a
⟨proof⟩

lemma invers2: [a ≤ b; (b−a) = p∗q] ⇒ (b::nat) = a+p∗q
⟨proof⟩

lemma modadd: [b = a+p∗q] ⇒ (a::nat) mod p = b mod p
⟨proof⟩

lemma equalmodstrick1: pdifference a b mod p = 0 ⇒ a mod p = b mod p
⟨proof⟩

lemma diff-add-assoc: b ≤ c ⇒ c − (c − b) = c − c + (b::nat)
⟨proof⟩

lemma diff-add-assoc2: a ≤ c ⇒ c − (c − a + b) = (c − c + (a::nat) − b)
⟨proof⟩

lemma diff-add-diff: x ≤ b ⇒ (b::nat) − x + y − b = y − x
⟨proof⟩

lemma equalmodstrick2:
  assumes a mod p = b mod p
  shows pdifference a b mod p = 0
⟨proof⟩

lemma primekeyrewrite:
  fixes p::nat shows [prime p; p dvd (a+b);¬(p dvd a)] ⇒ p dvd b
⟨proof⟩
lemma multzero: \[0 < m \mod p; m \ast a = 0\] \(\Rightarrow\) \((a::nat) = 0\)

\langle proof \rangle

lemma primekeytrick:
fixes \(A, B\) :: nat
assumes \((M \ast A) \mod P = (M \ast B) \mod P\)
assumes \(M \mod P \neq 0\) and prime \(P\)
shows \(A \mod P = B \mod P\)
\langle proof \rangle

end

7 Lemmata for modular arithmetic with primes

theory Productdivides
imports Pdifference
begin

lemma productdivides-lemma: \[x \mod z = (0::nat)\] \(\Rightarrow\) \(((y\ast x) \mod (y\ast z) = 0)\)

\langle proof \rangle

lemma productdivides: \[x \mod a = (0::nat); x \mod b = 0; \text{prime } a; \text{prime } b; a \neq b\] \(\Rightarrow\) \(x \mod (a\ast b) = 0\)

\langle proof \rangle

lemma specializedtoprimes1:
fixes \(p\) :: nat
shows \([\text{prime } p; \text{prime } q; p \neq q; a \mod p = b \mod p ; a \mod q = b \mod q]\)
\(\Rightarrow\) a mod \((p\ast q) = b \mod (p\ast q)\)

\langle proof \rangle

lemma specializedtoprimes1a:
fixes \(p\) :: nat
shows \([\text{prime } p; \text{prime } q; p \neq q; a \mod p = b \mod p ; a \mod q = b \mod q ; b < p\ast q ]\)
\(\Rightarrow\) a mod \((p\ast q) = b\)

\langle proof \rangle

end

8 Correctness proof for RSA

theory Cryptinverts
imports Crypt Productdivides ~~/src/HOL/Number-Theory/Residues
begin

In this theory we show, that a RSA encrypted message can be decrypted
primrec \textit{pred} :: \textit{nat} $\Rightarrow$ \textit{nat} \\
where \\
\textit{pred} 0 = 0 \\
\mid \textit{pred} \ (\textit{Suc} \ a) = a

\textbf{lemma \textit{fermat}:

\textbf{assumes prime} \ p \ m \ \mod \ p \neq 0 \\
\textbf{shows} \ m^{\left(\textit{p}-(1::\textit{nat})\right)} \mod \ p = 1

\langle \textit{proof} \rangle

\textbf{lemma \textit{cryptinverts-hilf1}:

\textbf{prime} \ p \implies (m \ast m^{\left(\textit{k} \ast (\textit{pred} \ p)\right)}) \mod \ p = m \mod \ p

\langle \textit{proof} \rangle

\textbf{lemma \textit{cryptinverts-hilf2}:

\textbf{prime} \ p \implies m\ast(m^{\left(\textit{k} \ast (\textit{pred} \ p) \ast (\textit{pred} \ q)\right)}) \mod \ p = m \mod \ p

\langle \textit{proof} \rangle

\textbf{lemma \textit{cryptinverts-hilf3}:

\textbf{prime} \ q \implies m\ast(m^{\left(\textit{k} \ast (\textit{pred} \ p) \ast (\textit{pred} \ q)\right)}) \mod \ q = m \mod \ q

\langle \textit{proof} \rangle

\textbf{lemma \textit{cryptinverts-hilf4}:

\left[ \textbf{prime} \ p; \textbf{prime} \ q; \textbf{p} \neq \textbf{q}; m < \textit{p} \ast \textit{q}; \textbf{x} \mod \ (\textit{pred} \ p)\ast(\textit{pred} \ q) = 1 \right] \implies m^{\textit{x}} \mod \ (\textit{p} \ast \textit{q}) = m

\langle \textit{proof} \rangle

\textbf{lemma \textit{primmultgreater}:

\textbf{fixes} \ p::\textit{nat} \textbf{ shows} \left[ \textbf{prime} \ p; \textbf{prime} \ q; \textbf{p} \neq 2; \textbf{q} \neq 2 \right] \implies 2 < \textit{p} \ast \textit{q}

\langle \textit{proof} \rangle

\textbf{lemma \textit{primmultgreater2}:

\textbf{fixes} \ p::\textit{nat} \textbf{ shows} \left[ \textbf{prime} \ p; \textbf{prime} \ q; \textbf{p} \neq \textbf{q} \right] \implies 2 < \textit{p} \ast \textit{q}

\langle \textit{proof} \rangle

\textbf{lemma \textit{cryptinverts}:

\left[ \textbf{prime} \ p; \textbf{prime} \ q; \textbf{p} \neq \textbf{q}; n = \textit{p} \ast \textit{q}; m < n; \textbf{e} \ast \textbf{d} \mod \ ((\textit{pred} \ p)\ast(\textit{pred} \ q)) = 1 \right] \implies \textbf{rsa-crypt} \ (\textbf{rsa-crypt} \ m \ e \ n) \ast \textbf{d} \ n = m

\langle \textit{proof} \rangle

\textbf{end}

9 Extensions to the Word theory required for PSS

theory \ \textit{Wordarith}
imports \ \textit{WordOperations} \sim/\textit{src}/\textit{HOL}/\textit{Number-Theory}/\textit{Primes}
begin

\textbf{definition}
\textit{nat-to-be-length} :: \textit{nat} $\Rightarrow$ \textit{nat} $\Rightarrow$ \textit{bv} \where

\textit{nat-to-be-length}:

\textbf{end}

10
\[ \text{nat-to-bv-length } n \; l = (\text{if } \text{length}(\text{nat-to-bv } n) \leq l \text{ then } \text{bv-extend } l \; 0 \text{ (nat-to-bv } n) \text{ else } []) \]

**Lemma** length-nat-to-bv-length:
\[ \text{nat-to-bv-length } x \; y \neq [] \implies \text{length (nat-to-bv-length } x \; y) = y \]
(\text{proof})

**Lemma** bv-to-nat-nat-to-bv-length:
\[ \text{nat-to-bv-length } x \; y \neq [] \implies \text{bv-to-nat (nat-to-bv-length } x \; y) = x \]
(\text{proof})

**Definition**
\[ \text{roundup :: nat } \Rightarrow \text{nat } \Rightarrow \text{nat where} \]
\[ \text{roundup } x \; y = (\text{if } (x \mod y = 0) \text{ then } (x \div y) \text{ else } (x \div y) + 1) \]

**Lemma** rndvdvd: \[ b \; \text{dvd } a \implies \text{roundup } a \; b \ast b = a \]
(\text{proof})

**Lemma** bv-to-nat-zero-prepend: \[ \text{bv-to-nat } a = \text{bv-to-nat } (0 \# a) \]
(\text{proof})

**Primrec** remzero:: \[ \text{bv } \Rightarrow \text{bv where} \]
\[ \text{remzero } [] = [] \]
\[ \text{remzero } (a \# b) = (\text{if } a = 1 \text{ then } (a \# b) \text{ else } \text{remzero } b) \]

**Lemma** remzeroeq: \[ \text{bv-to-nat } a = \text{bv-to-nat } (\text{remzero } a) \]
(\text{proof})

**Lemma** len-nat-to-bv-pos: \text{assumes } x: 1 < a \text{ shows } 0 < \text{length } (\text{nat-to-bv } a)
(\text{proof})

**Lemma** remzero-replicate: \[ \text{remzero } ((\text{replicate } n \; 0)@l) = \text{remzero } l \]
(\text{proof})

**Lemma** length-bvxor-bound: \[ a \leq \text{length } l \implies a \leq \text{length } (\text{bvxor } l \; l2) \]
(\text{proof})

**Lemma** nat-to-bv-helper-legacy-induct:
\[ (\forall n. \; n \neq (0::\text{nat}) \implies P \; (n \div 2) \implies P \; n) \implies P \; x \]
(\text{proof})

**Lemma** len-lower-bound:
\text{assumes } 0 < n \text{ shows } 2^\text{length } (\text{nat-to-bv } n) - \text{Suc } 0 \leq n
(\text{proof})
lemma length-lower: assumes a: length a < length b and b: (hd b) ~= 0 shows bv-to-nat a < bv-to-nat b
⟨proof⟩

lemma nat-to-bv-non-empty: assumes a: 0 < n shows nat-to-bv n ~= []
⟨proof⟩

lemma hd-append: x ~= [] ==> hd (x @ xs) = hd x
⟨proof⟩

lemma hd-one: 0 < n ==> hd (nat-to-bv-helper n []) = 1
⟨proof⟩

lemma prime-hd-non-zero:
  fixes p::nat assumes a: prime p and b: prime q shows hd (nat-to-bv (p*q)) ~= 0
⟨proof⟩

lemma primerew: fixes p::nat shows [ m dvd p; m~1; m~p] ==> ~ prime p
⟨proof⟩

lemma two-dvd-exp: 0<x ==> (2::nat) dvd 2^x
⟨proof⟩

lemma exp-prod1: [1<b; 2^x=2*(b::nat)] ==> 2 dvd b
⟨proof⟩

lemma exp-prod2: [1<a; 2^x=a*2] ==> (2::nat) dvd a
⟨proof⟩

lemma odd-mul-odd: [~(2::nat) dvd p; ~2 dvd q] ==> ~2 dvd p*q
⟨proof⟩

lemma prime-equal: fixes p::nat shows [prime p; prime q; 2^x=p*q] ==> (p=q)
⟨proof⟩

lemma nat-to-bv-length-bv-to-nat:
  length xs = n ==> xs ≠ [] ==> nat-to-bv-length (bv-to-nat xs) n = xs
⟨proof⟩

end

10 EMSA-PSS encoding and decoding operation

theory EMSAPSS
imports SHA1 Wordarith
begin
We define the encoding and decoding operations for the probabilistic signature scheme. Finally we show, that encoded messages always can be verified

```plaintext
definition show-rightmost-bits:: bv ⇒ nat ⇒ bv
  where show-rightmost-bits bvec n = rev (take n (rev bvec))

definition BC:: bv
  where BC = [One, Zero, One, One, One, Zero, Zero]

definition salt:: bv
  where salt = []

definition sLen:: nat
  where sLen = length salt

definition generate-M'::: bv ⇒ bv ⇒ bv
  where generate-M' mHash salt-new = bv-prepend 64 0 [] @ mHash @ salt-new

definition generate-PS:: nat ⇒ nat ⇒ bv
  where generate-PS emBits hLen = bv-prepend ((roundup emBits 8)*8 - sLen - hLen - 16) 0 []

definition generate-DB:: bv ⇒ bv
  where generate-DB PS = PS @ [Zero, Zero, Zero, Zero, Zero, Zero, Zero, One] @ salt

definition maskedDB-zero:: bv ⇒ nat ⇒ bv
  where maskedDB-zero maskedDB emBits = bv-prepend ((roundup emBits 8)*8 - emBits) 0 (drop ((roundup emBits 8)*8 - emBits) maskedDB)

definition generate-H:: bv ⇒ nat ⇒ nat ⇒ bv
  where generate-H EM emBits hLen = take hLen (drop ((roundup emBits 8)*8 - hLen - 8) EM)

definition generate-maskedDB:: bv ⇒ nat ⇒ nat ⇒ bv
  where generate-maskedDB EM emBits hLen = take ((roundup emBits 8)*8 - hLen - 8) EM

definition generate-salt:: bv ⇒ bv
  where generate-salt DB-zero = show-rightmost-bits DB-zero sLen

primrec MGF2:: bv ⇒ nat ⇒ bv
  where
    MGF2 Z 0 = sha1 (Z@(nat-to-bv-length 0 32))
  | MGF2 Z (Suc n) = (MGF2 Z n)@((sha1 (Z@(nat-to-bv-length (Suc n) 32)))

definition MGF1:: bv ⇒ nat ⇒ nat ⇒ bv
  where MGF1 Z n l = take l (MGF2 Z n)

definition MGF:: bv ⇒ nat ⇒ bv
```

13
where

\[ MGF \, Z \, l = (\text{if } l = 0 \lor 2^{32} \ast (\text{length } (\text{sha1 } Z)) < l \]
\[ \text{then } [] \]
\[ \text{else } MGF1 \, Z \, (\text{roundup } l \, (\text{length } (\text{sha1 } Z)) - 1 \, ) \]

\text{definition emsapss-encode-help8:: } bv \Rightarrow bv \Rightarrow bv
\text{where emsapss-encode-help8 } DBzero \, H = DBzero \, @ H \, @ BC

\text{definition emsapss-encode-help7:: } bv \Rightarrow bv \Rightarrow nat \Rightarrow bv
\text{where emsapss-encode-help7 } maskedDB \, H \, emBits =
\text{emsapss-encode-help8 } (\text{maskedDB-zero} \, \text{maskedDB} \, \text{emBits}) \, H

\text{definition emsapss-encode-help6:: } bv \Rightarrow bv \Rightarrow bv \Rightarrow nat \Rightarrow bv
\text{where emsapss-encode-help6 } DB \, dbMask \, H \, emBits =
\text{(if } dbMask = [] \text{ then } [] \text{ else emsapss-encode-help7 } (\text{bexor } DB \, dbMask) \, H \, emBits)

\text{definition emsapss-encode-help5:: } bv \Rightarrow bv \Rightarrow nat \Rightarrow bv
\text{where emsapss-encode-help5 } DB \, H \, emBits =
\text{emsapss-encode-help6 } DB \, (MGF \, H \, (\text{length } DB)) \, H \, emBits

\text{definition emsapss-encode-help4:: } bv \Rightarrow bv \Rightarrow nat \Rightarrow bv
\text{where emsapss-encode-help4 } PS \, H \, emBits =
\text{emsapss-encode-help5 } (\text{generate-DB } PS) \, H \, emBits

\text{definition emsapss-encode-help3:: } bv \Rightarrow nat \Rightarrow bv
\text{where emsapss-encode-help3 } H \, emBits =
\text{emsapss-encode-help4 } (\text{generate-PS } emBits \, (\text{length } H)) \, H \, emBits

\text{definition emsapss-encode-help2:: } bv \Rightarrow nat \Rightarrow bv
\text{where emsapss-encode-help2 } M' \, emBits = emsapss-encode-help3 \, (\text{sha1 } M') \, emBits

\text{definition emsapss-encode-help1:: } bv \Rightarrow nat \Rightarrow bv
\text{where emsapss-encode-help1 } mHash \, emBits =
\text{(if } emBits < \text{length } (mHash) + sLen + 16 \text{ then } [] \text{ else emsapss-encode-help2 } (\text{generate-M'} \, mHash \, \text{salt}) \, emBits)

\text{definition emsapss-encode:: } bv \Rightarrow nat \Rightarrow bv
\text{where emsapss-encode } M \, emBits =
\text{(if } 2^{64} \leq \text{length } M \lor 2^{32} \ast 160 < emBits \text{ then } [] \text{ else emsapss-encode-help1 } (\text{sha1 } M) \, emBits)

\text{definition emsapss-decode-help11:: } bv \Rightarrow bv \Rightarrow bool
where \texttt{emsapss-decode-help11} \( H' \) \( H = (\text{if } H' \neq H \text{ then False else True}) \)

definition \texttt{emsapss-decode-help10}:: \( bv \Rightarrow bv \Rightarrow bool \)
where \texttt{emsapss-decode-help10} \( M' \) \( H = \text{emsapss-decode-help11} \ (\text{sha1 } M') \ H \)

definition \texttt{emsapss-decode-help9}:: \( bv \Rightarrow bv \Rightarrow bool \)
where \texttt{emsapss-decode-help9} \( \text{mHash} \) \( \text{salt-new} \) \( H = \text{emsapss-decode-help10} \ (\text{generate-M} \ ' \ \text{mHash} \ \text{salt-new}) \ H \)

definition \texttt{emsapss-decode-help8}:: \( bv \Rightarrow bv \Rightarrow bool \)
where \texttt{emsapss-decode-help8} \( \text{mHash} \) \( \text{DB-zero} \) \( H = \text{emsapss-decode-help9} \ \text{mHash} \ (\text{generate-salt DB-zero}) \ H \)

definition \texttt{emsapss-decode-help7}:: \( bv \Rightarrow bv \Rightarrow bool \)
where \texttt{emsapss-decode-help7} \( \text{mHash} \) \( \text{DB-zero} \) \( H \) \( \text{emBits} \)
\( = \)
(\text{if} \ (\text{take} ((\text{roundup } \text{emBits} 8) \ast 8 - (\text{length } \text{mHash}) - \text{sLen} - 16) \ \text{DB-zero} \neq \ \text{bv-prepend} ((\text{roundup } \text{emBits} 8) \ast 8 - (\text{length } \text{mHash}) - \text{sLen} - 16) \ 0 \ []) \ \text{\lor} \ (\text{take} 8 (\text{drop} ((\text{roundup } \text{emBits} 8) \ast 8 - (\text{length } \text{mHash}) - \text{sLen} - 16) \ \text{DB-zero}) \neq [\text{Zero, Zero, Zero, Zero, Zero, Zero, Zero, One}]) \ \text{then False} \)
\text{else} \ \text{emsapss-decode-help8} \ \text{mHash} \ \text{DB-zero} \ H \)

definition \texttt{emsapss-decode-help6}:: \( bv \Rightarrow bv \Rightarrow bv \Rightarrow nat \Rightarrow bool \)
where \texttt{emsapss-decode-help6} \( \text{mHash} \) \( \text{DB-zero} \) \( \text{emBits} = \)
\( \text{emsapss-decode-help7} \ \text{mHash} \ \text{DB-zero} \ \text{emBits} = \)
(\text{if} \ (\text{take} ((\text{roundup } \text{emBits} 8) \ast 8 - (\text{length } \text{mHash}) - \text{sLen} - 16) \ \text{DB-zero} \neq \ \text{bv-prepend} ((\text{roundup } \text{emBits} 8) \ast 8 - (\text{length } \text{mHash}) - \text{sLen} - 16) \ 0 \ []) \ \text{\lor} \ (\text{take} 8 (\text{drop} ((\text{roundup } \text{emBits} 8) \ast 8 - (\text{length } \text{mHash}) - \text{sLen} - 16) \ \text{DB-zero}) \neq [\text{Zero, Zero, Zero, Zero, Zero, Zero, Zero, One}]) \ \text{then False} \)
\text{else} \ \text{emsapss-decode-help6} \ \text{mHash} \ \text{DB-zero} \ \text{emBits} \)

definition \texttt{emsapss-decode-help5}:: \( bv \Rightarrow bv \Rightarrow bv \Rightarrow nat \Rightarrow bool \)
where \texttt{emsapss-decode-help5} \( \text{mHash} \) \( \text{maskedDB} \) \( \text{dbMask} \) \( \text{emBits} = \)
\( \text{emsapss-decode-help6} \ \text{mHash} \ \text{maskedDB} \ \text{dbMask} \ \text{emBits} = \)
(\text{emsapss-decode-help6} \ \text{maskedDB} \ \text{dbMask} \ \text{emBits} \ \text{emBits} \)

definition \texttt{emsapss-decode-help4}:: \( bv \Rightarrow bv \Rightarrow bv \Rightarrow nat \Rightarrow bool \)
where \texttt{emsapss-decode-help4} \( \text{mHash} \) \( \text{maskedDB} \) \( \text{emBits} = \)
(\text{if} \ (\text{take} ((\text{roundup } \text{emBits} 8) \ast 8 - \text{emBits}) \ \text{maskedDB} \neq \ \text{bv-prepend} ((\text{roundup} \ \text{emBits} 8) \ast 8 - \text{emBits}) \ 0 \ []) \ \text{then False} \)
\text{else} \ \text{emsapss-decode-help4} \ \text{mHash} \ \text{maskedDB} \ \text{emBits} \)

definition \texttt{emsapss-decode-help3}:: \( bv \Rightarrow bv \Rightarrow nat \Rightarrow bool \)
where \texttt{emsapss-decode-help3} \( \text{mHash} \) \( \text{EM} \) \( \text{emBits} = \)
\( \text{emsapss-decode-help4} \ \text{mHash} \ \text{EM} \ \text{emBits} = \)
(\text{emsapss-decode-help4} \ \text{mHash} \ \text{EM} \ \text{emBits} \ \text{emBits} \)

definition \texttt{emsapss-decode-help2}:: \( bv \Rightarrow bv \Rightarrow nat \Rightarrow bool \)
where \texttt{emsapss-decode-help2} \( \text{mHash} \) \( \text{EM} \) \( \text{emBits} = \)
(\text{emsapss-decode-help2} \ \text{mHash} \ \text{EM} \ \text{emBits} = \)
(\text{emsapss-decode-help2} \ \text{mHash} \ \text{EM} \ \text{emBits} = \)

15
definition emsapss-decode-help1 :: bv ⇒ bv ⇒ nat ⇒ bool  
where emsapss-decode-help1 mHash EM emBits  
  (if emBits < length (mHash) + sLen + 16  
    then False  
    else emsapss-decode-help2 mHash EM emBits)  

definition emsapss-decode :: bv ⇒ bv ⇒ nat ⇒ bool  
where emsapss-decode M EM emBits  
  (if (2^64 ≤ length M ∨ 2^32*160<emBits)  
    then False  
    else emsapss-decode-help1 (sha1 M) EM emBits)  

lemma roundup-positive: 0 < emBits ⇒ 0 < (roundup emBits 160)  
⟨proof⟩

lemma roundup-ge-emBits: 0 < emBits ⇒ 0 < x ⇒ emBits ≤ (roundup emBits x) * x  
⟨proof⟩

lemma roundup-ge-0: 0 < emBits ⇒ 0 < x ⇒ 0 ≤ roundup emBits x * x – emBits  
⟨proof⟩

lemma roundup-le-7: 0 < emBits ⇒ roundup emBits 8 * 8 – emBits ≤ 7  
⟨proof⟩

lemma roundup-nat-ge-8-help:  
  length (sha1 M) + sLen + 16 ≤ emBits ⇒ 8 ≤ roundup emBits 8 * 8 – (length (sha1 M) + 8)  
⟨proof⟩

lemma roundup-nat-ge-8:  
  length (sha1 M) + sLen + 16 ≤ emBits ⇒ 8 ≤ roundup emBits 8 * 8 – (length (sha1 M) + 8)  
⟨proof⟩

lemma roundup-le-ub:  
  [ 176 + sLen ≤ emBits; emBits ≤ 2^32 * 160 ] ⇒ (roundup emBits 8) * 8 – 168 ≤ 2^32 * 160  
⟨proof⟩

lemma modify-roundup-g1: [8 ≤ roundup emBits 8 * 8 – 168] ⇒ 176 ≤ roundup emBits 8 * 8  
⟨proof⟩

lemma modify-roundup-g2: [176 ≤ roundup emBits 8 * 8] ⇒ 21 < roundup emBits 8  
⟨proof⟩

16
**Lemma** `roundup-help1`: \( [0 < \text{roundup } l \; 160] \implies (\text{roundup } l \; 160 - 1) + 1 = (\text{roundup } l \; 160) \)

(\text{proof})

**Lemma** `roundup-help1-new`: \( [0 < l] \implies (\text{roundup } l \; 160 - 1) + 1 = (\text{roundup } l \; 160) \)

(\text{proof})

**Lemma** `roundup-help2`: \([176 + sLen \leq \text{emBits}] \implies \text{roundup emBits } 8 \ast 8 - \text{emBits} \leq \text{roundup emBits } 8 \ast 8 - 160 - sLen - 16 \)

(\text{proof})

**Lemma** `bv-prepend-equal`: \( \text{bv-prepend } (\text{Suc } n) b \; l = b \# \text{bv-prepend } n b \; l \)

(\text{proof})

**Lemma** `length-bv-prepend`: \( \text{length } (\text{bv-prepend } n b \; l) = n + \text{length } l \)

(\text{proof})

**Lemma** `length-bv-prepend-drop`: \( a \leq \text{length } xs \implies \text{length } (\text{bv-prepend } a b \; (\text{drop } a \; xs)) = \text{length } xs \)

(\text{proof})

**Lemma** `take-bv-prepend`: \( \text{take } n (\text{bv-prepend } n b \; x) = \text{bv-prepend } n b \; [\] \)

(\text{proof})

**Lemma** `take-bv-prepend2`: \( \text{take } n (\text{bv-prepend } n b \; xs \; @ \; ys \; @ \; zs) = \text{bv-prepend } n b \; [\] \)

(\text{proof})

**Lemma** `bv-prepend-append`: \( \text{bv-prepend } a b \; x = \text{bv-prepend } a \; b \; [\] \; @ \; x \)

(\text{proof})

**Lemma** `bv-prepend-append2`: \( x < y \implies \text{bv-prepend } y b \; xs = (\text{bv-prepend } x b \; [\]) \; @ \; (\text{bv-prepend } (y - x) b \; [\]) \; @ \; xs \)

(\text{proof})

**Lemma** `drop-bv-prepend-help2`: \( [x < y] \implies \text{drop } x (\text{bv-prepend } y b \; [\]) = \text{bv-prepend } (y - x) b \; [\]

(\text{proof})

**Lemma** `drop-bv-prepend-help3`: \( [x = y] \implies \text{drop } x (\text{bv-prepend } y b \; [\]) = \text{bv-prepend } (y - x) b \; [\]

(\text{proof})

**Lemma** `drop-bv-prepend-help4`: \( [x \leq y] \implies \text{drop } x (\text{bv-prepend } y b \; [\]) = \text{bv-prepend } (y - x) b \; [\]

(\text{proof})

**Lemma** `bv-prepend-add`: \( \text{bv-prepend } x b \; [\] @ \text{bv-prepend } y b \; [\] = \text{bv-prepend } (x + y) b \; [\]

(\text{proof})

17
lemma bv-prepend-drop: \( x \leq y \rightarrow \text{bv-prepend } x \text{ b (drop } x \text{ (bv-prepend } y \text{ b []))} = \text{bv-prepend } y \text{ b []} \)

⟨proof⟩

lemma bv-prepend-split: \( \text{bv-prepend } x \text{ b (left } @ \text{ right)} = \text{bv-prepend } x \text{ b left } @ \text{ right} \)

⟨proof⟩

lemma length-generate-DB: \( \text{length (generate-DB PS)} = \text{length PS} + 8 + sLen \)

⟨proof⟩

lemma length-generate-PS: \( \text{length (generate-PS emBits 160)} = (\text{roundup emBits 8}) * 8 - sLen - 160 - 16 \)

⟨proof⟩

lemma length-bvxor: \( \text{length a = length b} \implies \text{length (bvxor a b)} = \text{length a} \)

⟨proof⟩

lemma length-MGF2: \( \text{length (MGF2 Z m)} = \text{Suc m} * \text{length (sha1 (Z } @ \text{ nat-to-bv-length m 32))} \)

⟨proof⟩

lemma length-MGF1: \( l \leq (\text{Suc n} ) * 160 \implies \text{length (MGF1 Z n l)} = l \)

⟨proof⟩

lemma length-MGF: \( 0 < l \implies l \leq 2^32 * \text{length (sha1 x)} \implies \text{length (MGF x l)} = l \)

⟨proof⟩

lemma solve-length-generate-DB:
\[
[ 0 < \text{emBits}; \text{length (sha1 M)} + sLen + 16 \leq \text{emBits} ] \\
\implies \text{length (generate-DB (generate-PS emBits (length (sha1 x))) )} = (\text{roundup emBits 8}) * 8 - 168
\]

⟨proof⟩

lemma length-maskedDB-zero:
\[
[ \text{roundup emBits 8 } * \text{8 } - \text{emBits} \leq \text{length maskedDB} ] \\
\implies \text{length (maskedDB-zero maskedDB emBits) = length maskedDB}
\]

⟨proof⟩

lemma take-equal-bv-prepend:
\[
[ 176 + sLen \leq \text{emBits}; \text{roundup emBits 8 } * \text{8 } - \text{emBits} \leq 7 ] \\
\implies \text{take (roundup emBits 8 } * \text{8 } - \text{length (sha1 M)} - sLen - 16) \text{ (maskedDB-zero (generate-DB (generate-PS emBits 160)) emBits) = } \\
\text{bv-prepend (roundup emBits 8 } * \text{8 } - \text{length (sha1 M)} - sLen - 16) \text{ 0 []}
\]

⟨proof⟩

lemma lastbits-BC: \( BC = \text{show-rightmost-bits (xs } @ \text{ ys } @ \text{ BC)} 8 \)

18
proof

lemma equal-zero:
176 + sLen ≤ emBits ⇒ roundup emBits 8 * 8 - emBits ≤ roundup emBits 8 * 8 - (176 + sLen)
⇒ 0 = roundup emBits 8 * 8 - emBits - (roundup emBits 8 * 8 - (176 + sLen))
(proof)

lemma get-salt: [ 176 + sLen ≤ emBits; roundup emBits 8 * 8 - emBits ≤ 7 ] ⇒ (generate-salt (maskedDB-zero (generate-DB (generate-PS emBits 160))) emBits)) = salt
(proof)

lemma generate-maskedDB-elim: [roundup emBits 8 * 8 - emBits ≤ length x; ( roundup emBits 8 ) * 8 - (length (sha1 M) - 8 = length (maskedDB-zero x emBits) (length(sha1 M)) = maskedDB-zero x emBits
(proof)

lemma generate-H-elim: [roundup emBits 8 * 8 - emBits ≤ length x; length (maskedDB-zero x emBits) = (roundup emBits 8) * 8 - 168; length y = 160] ⇒ generate-H (maskedDB-zero x emBits @ y @ z) emBits 160 = y
(proof)

lemma length-bv-prepend-drop-special: []roundup emBits 8 * 8 - emBits <= length x; length (generate-PS emBits 160) = roundup emBits 8 * 8 - (176 + sLen)] => length ( bv-prepend (roundup emBits 8 * 8 - emBits) 0 (drop (roundup emBits 8 * 8 - emBits) (generate-PS emBits 160))) = length (generate-PS emBits 160)
(proof)

lemma x01-elim: [176 + sLen ≤ emBits; roundup emBits 8 * 8 - emBits ≤ 7] => take 8 (drop (roundup emBits 8 * 8 - (length (sha1 M) + sLen + 16))(maskedDB-zero (generate-DB (generate-PS emBits 160)) emBits)) = [0, 0, 0, 0, 0, 0, 0, 1]
(proof)

lemma drop-bv-mapzip:
assumes n <= length x length x = length y
shows drop n (bv-mapzip f x y) = bv-mapzip f (drop n x) (drop n y)
(proof)

lemma [simp]:
assumes length a = length b
shows bexor (bexor a b) b = a
(proof)

lemma bexorxor-elim-help:

19
assumes $x \leq \text{length } a \text{ and } \text{length } a = \text{length } b$
shows $\text{bv-prepend } x \ 0 \ (\text{drop } x \ (\text{bv-prepend } x \ 0 \ (\text{drop } x \ (\text{bv-or } a \ b)))) b)$

$= \text{bv-prepend } x \ 0 \ (\text{drop } x \ a)$

(proof)

lemma $\text{bv-orxor-elim}$: \[
\begin{array}{l}
\text{roundup emBits } 8 * 8 - \text{emBits} \leq \text{length } a; \text{length } a = \text{length } b \\
\implies (\text{maskedDB-zero} \ (\text{maskedDB-zero} \ (\text{bv-or } a \ b) \ \text{emBits}) \ b) \\
\text{emBits} = \text{bv-prepend} \ (\text{roundup emBits } 8 * 8 - \text{emBits}) \ 0 \ (\text{drop} \ (\text{roundup emBits } 8 * 8 - \text{emBits}) \ a)
\end{array}
\]

(proof)

lemma $\text{verify}$: \[
\begin{array}{l}
((\text{emsapss-encode } M \ \text{emBits}) \neq []; \ EM = (\text{emsapss-encode } M \ \text{emBits})) \implies \text{emsapss-decode } M \ EM \ \text{emBits} = \text{True}
\end{array}
\]

(proof)

end

11 RSS-PSS encoding and decoding operation

theory RSAPSS
imports EMSAPSS Cryptinverts
begin

We define the RSA-PSS signature and verification operations. Moreover we show, that messages signed with RSA-PSS can always be verified

definition $\text{rsapss-sign-help1}$:: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bv}$
where $\text{rsapss-sign-help1 } \text{em-nat } e \ n = \text{nat-to-bv-length} \ (\text{rsa-crypt } \text{em-nat } e \ n) \ (\text{length } \text{nat-to-bv } n))$

definition $\text{rsapss-sign}$:: $\text{bv} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bv}$
where $\text{rsapss-sign } m \ e \ n = \text{if } (\text{emsapss-encode } m \ (\text{length } \text{nat-to-bv } n) - 1)) = [] \text{ then } [] \text{ else } (\text{rsapss-sign-help1 } (\text{bv-to-nat} \ (\text{emsapss-encode } m \ (\text{length } \text{nat-to-bv } n) - 1))) \ e \ n))$

definition $\text{rsapss-verify}$:: $\text{bv} \Rightarrow \text{bv} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$
where $\text{rsapss-verify } m \ s \ d \ n = \text{if } (\text{length } s) \neq \text{length}(\text{nat-to-bv } n) \text{ then False } \text{ else let } \text{em} = \text{nat-to-bv-length} \ (\text{rsa-crypt } (\text{bv-to-nat } s) \ d \ n) \ ((\text{roundup} \ (\text{length}(\text{nat-to-bv } n) - 1)) \ 8) * 8) \ \text{in } \text{emsapss-decode } m \ \text{em} \ (\text{length}(\text{nat-to-bv } n) - 1))$

lemma $\text{length-emsapss-encode}$:
$\text{emsapss-encode } m \ x \neq [] \implies \text{length } (\text{emsapss-encode } m \ x) = \text{roundup } x \ 8 \ * 8$

(proof)
lemma bv-to-nat-emsapss-encode-le: emsapss-encode m x ≠ [] → bv-to-nat (emsapss-encode m x) < 2^8 (roundup x 8 * 8)
⟨proof⟩

lemma length-helper1: shows length
  ⟨bezor
  (generate-DB
   (generate-PS (length (nat-to-bv (p * q)) − Suc 0)
   (length (sha1 (generate-M'(sha1 m) salt))))))
  (MGF (sha1 (generate-M'(sha1 m) salt)))
  (length
   (generate-DB
    (generate-PS (length (nat-to-bv (p * q)) − Suc 0)
     (length (sha1 (generate-M'(sha1 m) salt)))))) @ sha1 (generate-M'(sha1 m) salt) @ BC)
  = length
  ⟨bezor
  (generate-DB
   (generate-PS (length (nat-to-bv (p * q)) − Suc 0)
   (length (sha1 (generate-M'(sha1 m) salt))))))
  (MGF (sha1 (generate-M'(sha1 m) salt)))
  (length
   (generate-DB
    (generate-PS (length (nat-to-bv (p * q)) − Suc 0)
     (length (sha1 (generate-M'(sha1 m) salt)))))) @ sha1 (generate-M'(sha1 m) salt) @ BC)
  = length
  ⟨bezor
  (generate-DB
   (generate-PS (length (nat-to-bv (p * q)) − Suc 0)
   (length (sha1 (generate-M'(sha1 m) salt))))))
  (MGF (sha1 (generate-M'(sha1 m) salt)))
  (length
   (generate-DB
    (generate-PS (length (nat-to-bv (p * q)) − Suc 0)
     (length (sha1 (generate-M'(sha1 m) salt)))))) + 168
  ⟩
⟨proof⟩

lemma MGFLen-helper: MGF z l ≈ [] → l <= 2^32*(length (sha1 z))
⟨proof⟩

lemma length-helper2: assumes p: prime p and q: prime q and mfg: (MGF (sha1 (generate-M'(sha1 m) salt)))
  (length
   (generate-DB
    (generate-PS (length (nat-to-bv (p * q)) − Suc 0)
     (length (sha1 (generate-M'(sha1 m) salt))))))
  (MGF (sha1 (generate-M'(sha1 m) salt)))
  (length
   (generate-DB
    (generate-PS (length (nat-to-bv (p * q)) − Suc 0)
     (length (sha1 (generate-M'(sha1 m) salt))))))
  and len: length (sha1 M) + sLen + 16 ≤ (length (nat-to-bv (p * q))) − Suc 0
  shows length
  (⟨bezor
  (generate-DB
   (generate-PS (length (nat-to-bv (p * q)) − Suc 0)
    (length (sha1 (generate-M'(sha1 m) salt))))))
  (MGF (sha1 (generate-M'(sha1 m) salt)))
  (length
   (generate-DB
    (generate-PS (length (nat-to-bv (p * q)) − Suc 0)
     (length (sha1 (generate-M'(sha1 m) salt)))))))
  ) = (roundup (length (nat-to-bv (p * q)) − Suc 0) 8) * 8 * 168
21
lemma emBits-roundup-cancel: \( emBits \mod 8 \sim 0 \Rightarrow \text{roundup} \ emBits \ 8 - \ emBits = 8 - (emBits \mod 8) \)

lemma emBits-roundup-cancel2: \( emBits \mod 8 \sim 0 \Rightarrow \text{roundup} \ emBits \ 8 - (8 - (emBits \mod 8)) = emBits \)

lemma length-bound: \[ \text{assumes p: prime q and} \ x: (length \text{nat-to-bv} (p \ast q) - \text{Suc 0}) \]

lemma length-bound2: \( 8 \leq (\text{length} (\text{nat-to-bv} (p \ast q)) - \text{Suc 0}) \mod 8 \sim 0; 8 \leq \text{length} (\text{maskedDB}) \Rightarrow \text{length} (\text{remzero} ((\text{maskedDB-zero} \text{maskedDB} \text{emBits} @ a @ b)) <= \text{length} (\text{maskedDB} @ a @ b) - (8 - (emBits \mod 8)) \)

lemma length-helper: \[ \text{assumes p: prime q and} \ x: (length \text{nat-to-bv} (p \ast q) - \text{Suc 0}) \mod 8 \sim 0 \text{ and mgf: } (\text{MGF} (\text{sha1} (\text{generate-M'} (\text{sha1} m) \text{salt})))) \sim [] \]

and len: \( \text{length} (\text{sha1} M) + sLen + 16 \leq (\text{length} (\text{nat-to-bv} (p \ast q))) - \text{Suc 0} \)

shows \( \text{length} (\text{remzero}) \)

(\text{maskedDB-zero} \text{bexor} \)

(\text{generate-DB} \text{generate-PS} (length \text{nat-to-bv} (p \ast q)) - \text{Suc 0}) \text{length} (\text{sha1} (\text{generate-M'} (\text{sha1} m) \text{salt}))) \text{MGF} (\text{sha1} (\text{generate-M'} (\text{sha1} m) \text{salt}))) \text{length} (\text{generate-DB} \text{generate-PS} (length \text{nat-to-bv} (p \ast q)) - \text{Suc 0}) \text{length} (\text{sha1} (\text{generate-M'} (\text{sha1} m) \text{salt}))) \sim [] \)

22
(proof)

lemma length-emsapss-smaller-pq: \[ \text{prime } p; \text{prime } q; \text{emsapss-encode } m (\text{length (nat-to-bv } (p \ast q) \text{) } - \text{Suc 0}) \neq []; (\text{length (nat-to-bv } (p \ast q) \text{) } - \text{Suc 0}) \text{ mod 8 } \sim= 0] \implies \text{length (remzero (emsapss-encode } m (\text{length (nat-to-bv } (p \ast q) \text{) } - \text{Suc 0})) < \text{length (nat-to-bv } (p \ast q)) \]

(\proof)

lemma bv-to-nat-emsapss-smaller-pq: \text{assumes } a: \text{prime } p \text{ and } b: \text{prime } q \text{ and } c: \text{emsapss-encode } m (\text{length (nat-to-bv } (p \ast q) \text{) } - \text{Suc 0}) \neq []; \text{shows } \text{bv-to-nat (emsapss-encode } m (\text{length (nat-to-bv } (p \ast q) \text{) } - \text{Suc 0})) < p \ast q \]

(\proof)

lemma rsa-pss-verify: \[ \text{prime } p; \text{prime } q; p \neq q; n = p \ast q; e \ast d \text{ mod ((pred } p) \ast (\text{pred } q)) = 1; \text{rsapss-sign } m e n \neq []; s = \text{rsapss-sign } m e n \implies \text{rsapss-verify } m s d n = \text{True} \]

(\proof)

end

References


