

# RSAPSS

Christina Lindenberg and Kai Wirt  
Darmstadt Technical University  
Cryptography and Computeralgebra

December 12, 2009

## 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

<b>1</b>	<b>Extensions to the Isabelle Word theory required for SHA1</b>	<b>2</b>
<b>2</b>	<b>Message Padding for SHA1</b>	<b>5</b>
<b>3</b>	<b>Formal definition of the secure hash algorithm</b>	<b>6</b>
<b>4</b>	<b>Definition of rsacrypt</b>	<b>9</b>
<b>5</b>	<b>Leammata for modular arithmetic</b>	<b>9</b>
<b>6</b>	<b>Positive differences</b>	<b>10</b>
<b>7</b>	<b>Lemmata for modular arithmetic with primes</b>	<b>12</b>
<b>8</b>	<b>Pigeon hole principle</b>	<b>13</b>
<b>9</b>	<b>Fermats little theorem</b>	<b>21</b>
<b>10</b>	<b>Correctness proof for RSA</b>	<b>24</b>
<b>11</b>	<b>Extensions to the Word theory required for PSS</b>	<b>26</b>
<b>12</b>	<b>EMSA-PSS encoding and decoding operation</b>	<b>33</b>

## 1 Extensions to the Isabelle Word theory required for SHA1

```
theory WordOperations
imports Word
begin
```

```
types bv = bit list
```

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

**definition**

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

**definition**

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

**definition**

```
bvor: vor 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 bvro1 where**

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

**fun bvr0r where**

```
bvr0r a 0 = a
| bvr0r [] x = []
| bvr0r x (Suc n) = bvr0r (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**

```

addmod32: addmod32 a b =
  rev (select (rev (nat-to-bv ((bv-to-nat a) + (bv-to-nat b)))) 0 31)

```

**definition**

```

bv-prepend: bv-prepend x b bv = replicate x b @ bv

```

**primrec zerolist where**

```

zerolist 0 = []
| zerolist (Suc n) = zerolist n @ [Zero]

```

**primrec hextovb where**

```

hextovb x0 = [Zero,Zero,Zero,Zero]
| hextovb x1 = [Zero,Zero,Zero,One]
| hextovb x2 = [Zero,Zero,One,Zero]
| hextovb x3 = [Zero,Zero,One,One]
| hextovb x4 = [Zero,One,Zero,Zero]
| hextovb x5 = [Zero,One,Zero,One]
| hextovb x6 = [Zero,One,One,Zero]
| hextovb x7 = [Zero,One,One,One]
| hextovb x8 = [One,Zero,Zero,Zero]
| hextovb x9 = [One,Zero,Zero,One]
| hextovb xA = [One,Zero,One,Zero]
| hextovb xB = [One,Zero,One,One]
| hextovb xC = [One,One,Zero,Zero]
| hextovb xD = [One,One,Zero,One]
| hextovb xE = [One,One,One,Zero]
| hextovb xF = [One,One,One,One]

```

**primrec hexvtobv where**

```

hexvtobv [] = []
| hexvtobv (x#r) = hextovb x @ hexvtobv r

```

**lemma selectlenhelp:**  $\text{length (selecthelp l i)} = (i + 1)$

**proof** (induct i arbitrary: l)

**case 0**

**show**  $\text{length (selecthelp l 0)} = 0 + 1$

**proof** (cases l)

**case Nil**

**then have**  $\text{selecthelp l 0} = [\text{Zero}]$  **by simp**

**then show** *?thesis* **by simp**

**next**

**case** (Cons a as)

**then have**  $\text{selecthelp l 0} = [a]$  **by simp**

**then show** *?thesis* **by simp**

```

qed
next
case (Suc x)
show length (selecthelp l (Suc x)) = (Suc x) + 1
proof (cases l)
  case Nil
  then have selecthelp l (Suc x) = Zero # selecthelp l x by simp
  then show length (selecthelp l (Suc x)) = Suc x + 1 using Suc by simp
next
case (Cons a b)
then have selecthelp l (Suc x) = a # selecthelp b x by simp
then have length (selecthelp l (Suc x)) = 1 + length (selecthelp b x) by simp
then show length (selecthelp l (Suc x)) = Suc x + 1 using Suc by simp
qed
qed

lemma selectlenhelp2:  $\bigwedge i. \text{ALL } l j. \text{EX } k. \text{select } l i j = \text{select } k 0 (j - i)$ 
proof (auto)
  fix i
  show  $\bigwedge l j. \exists k. \text{select } l i j = \text{select } k 0 (j - i)$ 
  proof (induct i)
    fix l and j
    have  $\text{select } l 0 j = \text{select } l 0 (j - (0::\text{nat}))$  by simp
    then show  $\text{EX } k. \text{select } l 0 j = \text{select } k 0 (j - (0::\text{nat}))$  by auto
  next
    case (Suc x)
    have b:  $\text{select } l (Suc x) j = \text{select } (tl l) x (j - (1::\text{nat}))$ 
    proof (cases l)
      case Nil
      then have  $\text{select } l (Suc x) j = \text{select } l x (j - (1::\text{nat}))$  by simp
      moreover have  $tl l = l$  using Nil by simp
      ultimately show ?thesis by (simp)
    next
      case (Cons head tail)
      then have  $\text{select } l (Suc x) j = \text{select } tail x (j - (1::\text{nat}))$  by simp
      moreover have  $tail = tl l$  using Cons by simp
      ultimately show ?thesis by simp
    qed
    have  $\exists k. \text{select } l x j = \text{select } k 0 (j - (x::\text{nat}))$  using Suc by simp
    moreover have  $\text{EX } k. \text{select } (tl l) x (j - (1::\text{nat})) = \text{select } k 0 (j - (1::\text{nat}) - (x::\text{nat}))$ 
  using Suc[of  $tl l j - (1::\text{nat})$ ] by auto
  ultimately have  $\text{EX } k. \text{select } l (Suc x) j = \text{select } k 0 (j - (1::\text{nat}) - (x::\text{nat}))$ 
  using b by auto
  then show  $\text{EX } k. \text{select } l (Suc x) j = \text{select } k 0 (j - Suc x)$  by simp
  qed
  qed
qed

lemma selectlenhelp3:  $\text{ALL } j. \text{select } l 0 j = \text{selecthelp } l j$ 
proof

```

```

fix j
show select l 0 j = selecthelp l j
proof (cases l)
  case Nil
    assume l=[]
    then show select l 0 j = selecthelp l j by simp
  next
    case (Cons a b)
    then show select l 0 j = selecthelp l j by simp
qed
qed

```

```

lemma selectlen: length (select l i j) = j - i + 1
proof -
  from selectlenhelp2 have EX k. select l i j = select k 0 (j-i) by simp
  then have EX k. length (select l i j) = length (select k 0 (j-i)) by auto
  then have c: EX k. length (select l i j) = length (selecthelp k (j-i))
    using selectlenhelp3 by simp
  from c obtain k where d: length (select l i j) = length (selecthelp k (j-i)) by
  auto
  have 0 <= j-i by arith
  then have length (selecthelp k (j-i)) = j-i+1 using selectlenhelp by simp
  then show length (select l i j) = j-i+1 using d by simp
qed

```

```

lemma addmod32len:  $\bigwedge$  a b. length (addmod32 a b) = 32
  using selectlen [of - 0 31] addmod32 by simp

```

**end**

## 2 Message Padding for SHA1

```

theory SHA1Padding
imports WordOperations
begin

```

```

definition zerocount :: nat  $\Rightarrow$  nat where
  zerocount: zerocount n = (((n + 64) div 512) + 1) * 512 - n - (65::nat)

```

```

definition helppadd :: bv  $\Rightarrow$  bv  $\Rightarrow$  nat  $\Rightarrow$  bv where
  helppadd x y n = x @ [One] @ zerolist (zerocount n) @ zerolist (64 - length y)
  @y

```

```

definition sha1padd :: bv  $\Rightarrow$  bv where
  sha1padd: sha1padd x = helppadd 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** *fif* **where**

*fif*:  $fif\ x\ y\ z = bvor\ (bvand\ x\ y)\ (bvand\ (bv-not\ x)\ z)$

**definition** *fxor* **where**

*fxor*:  $fxor\ x\ y\ z = bvxor\ (bvxor\ x\ y)\ z$

**definition** *fmaj* **where**

*fmaj*:  $fmaj\ x\ y\ z = bvor\ (bvor\ (bvand\ x\ y)\ (bvand\ x\ z))\ (bvand\ y\ z)$

**definition** *fselect* ::  $nat \Rightarrow bit\ list \Rightarrow bit\ list \Rightarrow bit\ list \Rightarrow bit\ list$  **where**

*fselect*:  $fselect\ r\ x\ y\ z = (if\ (r < 20)\ then\ (fif\ x\ y\ z)\ else$   
 $(if\ (r < 40)\ then\ (fxor\ x\ y\ z)\ else$   
 $(if\ (r < 60)\ then\ (fmaj\ x\ y\ z)\ else$   
 $(fxor\ x\ y\ z)))$

**definition** *K1* **where**

*K1*:  $K1 = hexvtobv\ [x5, xA, x8, x2, x7, x9, x9, x9]$

**definition** *K2* **where**

*K2*:  $K2 = hexvtobv\ [x6, xE, xD, x9, xE, xB, xA, x1]$

**definition** *K3* **where**

*K3*:  $K3 = hexvtobv\ [x8, xF, x1, xB, xB, xC, xD, xC]$

**definition** *K4* **where**

*K4*:  $K4 = hexvtobv\ [xC, xA, x6, x2, xC, x1, xD, x6]$

**definition** *kselect* ::  $nat \Rightarrow bit\ list$  **where**

*kselect*:  $kselect\ r = (if\ (r < 20)\ then\ K1\ else$   
 $(if\ (r < 40)\ then\ K2\ else$   
 $(if\ (r < 60)\ then\ K3\ else$   
 $K4)))$

**definition** *getblock* **where**

*getblock*:  $getblock\ x = select\ x\ 0\ 511$

**fun** *delblockhelp* **where**

*delblockhelp* []  $n = []$   
| *delblockhelp* ( $x\#\#r$ )  $n = (if\ n \leq 0\ then\ x\#\#r\ else\ delblockhelp\ r\ (n - (1::nat)))$

**definition** *delblock* **where**

*delblock*: *delblock*  $x = \text{delblockhelp } x \ 512$

**primrec** *sha1compress* **where**

```

sha1compress 0 b A B C D E = (let j = (79::nat) in
  (let W = select b (32*j) ((32*j)+31) in
    (let AA = addmod32 (addmod32 (addmod32 W
      (bvrol A 5)) (fselect j B C D)) (addmod32 E (kselect j));
      BB = A;
      CC = bvrol B 30;
      DD = C;
      EE = D in
        AA@BB@CC@DD@EE)))
| sha1compress (Suc n) b A B C D E = (let j = (79 - (Suc n)) in
  (let W = select b (32*j) ((32*j)+31) in
    (let AA = addmod32 (addmod32 (addmod32 W
      (bvrol A 5)) (fselect j B C D)) (addmod32 E (kselect j));
      BB = A;
      CC = bvrol B 30;
      DD = C;
      EE = D in
        sha1compress n b AA BB CC DD EE)))

```

**definition** *sha1expandhelp* **where**

```

sha1expandhelp x i = (let j = (79+16-i) in (bvrol (bxor(bxor(
  select x (32*(j-(3::nat))) (31+(32*(j-(3::nat)))))) (select x (32*(j-(8::nat)))
  (31+(32*(j-(8::nat)))))) (bxor(select x (32*(j-(14::nat))) (31+(32*(j-(14::nat))))))
  (select x (32*(j-(16::nat))) (31+(32*(j-(16::nat)))))) 1))

```

**fun** *sha1expand* **where**

```

sha1expand x i = (if (i < 16) then x else
  let y = sha1expandhelp x i in
  sha1expand (x @ y) (i - 1))

```

**definition** *sha1compressstart* **where**

```

sha1compressstart: 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@BB@CC@DD@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);

```

$DD = \text{addmod32 } D \text{ (select H 96 127);}$   
 $EE = \text{addmod32 } E \text{ (select H 128 159)}$   
*in sha1block (getblock x) (delblock x) AA BB CC DD E)*

by *pat-completeness auto*

**termination proof** –

**have**  $aux: \bigwedge n \, xs :: \text{bit list. length (delblockhelp xs n)} \leq \text{length xs}$   
**proof** –  
    **fix**  $n$  **and**  $xs :: \text{bit list}$   
    **show**  $\text{length (delblockhelp xs n)} \leq \text{length xs}$   
    by (*induct n rule: delblockhelp.induct*) *auto*  
**qed**  
**have**  $\bigwedge x \, xs :: \text{bit list. length (delblock (x\#xs))} < \text{Suc (length xs)}$   
**proof** –  
    **fix**  $x$  **and**  $xs :: \text{bit list}$   
    **from**  $aux$  **have**  $\text{length (delblockhelp xs 511)} < \text{Suc (length xs)}$   
    **using** *le-less-trans [of length (delblockhelp xs 511) length xs]* **by** *auto*  
    **then show**  $\text{length (delblock (x\#xs))} < \text{Suc (length xs)}$  **by** (*simp add: delblock*)  
**qed**  
**then show** *?thesis*  
    by (*relation measure* ( $\lambda(b, x, A, B, C, D, E). \text{length } x$ )) *auto*  
**qed**

**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**

$sha1: sha1 \, x = (\text{let } y = \text{sha1padd } x \text{ in}$   
 $sha1block \text{ (getblock } y) \text{ (delblock } y) \, IV1 \, IV2 \, IV3 \, IV4 \, IV5)$

**lemma sha1blocklen:**  $\text{length (sha1block b x A B C D E)} = 160$

**proof** (*induct b x A B C D E rule: sha1block.induct*)

**case 1 show** *?case* **by** (*simp add: Let-def addmod32len*)

**next**

**case 2 then show** *?case* **by** (*simp add: Let-def*)

**qed**

**lemma** *sha1len*:  $\text{length } (\text{sha1 } m) = 160$   
**unfolding** *sha1* *Let-def sha1blocklen* ..

**end**

## 4 Definition of rsacrypt

**theory** *Crypt*  
**imports** *Mod*  
**begin**

This theory defines the rsacrypt function which implements RSA using fast exponentiation. An proof, that this function calculates RSA is also given

**definition**

*even* ::  $\text{nat} \Rightarrow \text{bool}$  **where**  
*even*  $n \longleftrightarrow 2 \text{ dvd } n$

**fun** *rsa-crypt* ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$   
**where**

*rsa-crypt*  $M$  0  $n = 1$   
| *rsa-crypt*  $M$  (*Suc*  $e$ )  $n = (\text{if } \text{even } (\text{Suc } e)$   
     $\text{then } (\text{rsa-crypt } M (\text{Suc } e \text{ div } 2) n) ^ 2 \text{ mod } n$   
     $\text{else } (M * ((\text{rsa-crypt } M (\text{Suc } e \text{ div } 2) n) ^ 2 \text{ mod } n)) \text{ mod } n)$

**lemma** *div-2-times-2*:  $(\text{if } (\text{even } m) \text{ then } (m \text{ div } 2 * 2 = m) \text{ else } (m \text{ div } 2 * 2 = m - 1))$

**by** (*simp add: even-def dvd-eq-mod-eq-0 mult-commute mult-div-cancel*)

**theorem** *cryptcorrect*:  $n \neq 0 \Longrightarrow n \neq 1 \Longrightarrow \text{rsa-crypt } M e n = M ^ e \text{ mod } n$

**by** (*induct*  $M$   $e$   $n$  *rule: rsa-crypt.induct*)  
(*auto simp add: power-mult [symmetric] div-2-times-2 remainderexp timesmod1*)

**end**

## 5 Lemmata for modular arithmetic

**theory** *Mod*  
**imports** *Main*  
**begin**

**lemma** *divmultassoc*:  $a \text{ div } (b*c) * (b*c) = ((a \text{ div } (b * c)) * b)*(c::\text{nat})$   
**by** *auto*

**lemma** *delmod*:  $(a::\text{nat}) \text{ mod } (b*c) \text{ mod } c = a \text{ mod } c$   
**apply** (*subst* (2) *mod-div-equality [symmetric, of a b\*c]*)

```

apply (subst add-commute)
apply (subst divmultassoc)
apply (simp add: mod-mult-self1)
done

lemma timesmod1:  $((x::nat)*(y::nat) \bmod n) \bmod (n::nat) = ((x*y) \bmod n)$ 
apply (subst mod-mult-distrib2)
apply (subst delmod)
apply auto
done

lemma timesmod3:  $((a \bmod (n::nat)) * b) \bmod n = (a*b) \bmod n$ 
apply (subst mult-commute)
apply (subst timesmod1)
apply (subst mult-commute)
apply auto
done

lemma remainderexplemma:  $(y \bmod (a::nat) = z \bmod a) \implies (x*y) \bmod a = (x*z)$ 
mod a
apply (subst timesmod1 [symmetric])
apply auto
apply (subst timesmod1)
apply auto
done

lemma remainderexp:  $((a \bmod (n::nat))^i) \bmod n = (a^i) \bmod n$ 
apply (induct i)
apply auto
apply (subst timesmod3)
apply (rule remainderexplemma)
apply auto
done

end

```

## 6 Positive differences

```

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

definition
  pdifference :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat where
    [simp]: pdifference a b = (if a < b then (b-a) else (a-b))

lemma timesdistributesoverpdifference:
   $m*(pdifference a b) = pdifference (m*(a::nat)) (m*(b::nat))$ 

```

```

by (auto simp add: nat-distrib)

lemma addconst:  $a = (b::nat) \implies c+a = c+b$ 
  by auto

lemma invers:  $a \leq x \implies (x::nat) = x - a + a$ 
  by auto

lemma invers2:  $\llbracket a \leq b; (b-a) = p*q \rrbracket \implies (b::nat) = a+p*q$ 
  apply (subst addconst [symmetric])
  apply (assumption)
  apply (subst add-commute, rule invers, simp)
  done

lemma modadd:  $\llbracket b = a+p*q \rrbracket \implies (a::nat) \bmod p = b \bmod p$ 
  by auto

lemma equalmodstrick1:  $pdifference\ a\ b\ \bmod\ p = 0 \implies a \bmod p = b \bmod p$ 
  apply (cases  $a < b$ )
  apply auto
  apply (rule modadd, rule invers2, auto)
  apply (rule sym)
  apply (rule modadd, rule invers2, auto)
  done

lemma diff-add-assoc:  $b \leq c \implies c - (c - b) = c - c + (b::nat)$ 
  by auto

lemma diff-add-assoc2:  $a \leq c \implies c - (c - a + b) = (c - c + (a::nat) - b)$ 
  apply (subst diff-diff-left [symmetric])
  apply (subst diff-add-assoc)
  apply auto
  done

lemma diff-add-diff:  $x \leq b \implies (b::nat) - x + y - b = y - x$ 
  by (induct b) auto

lemma equalmodstrick2:  $a \bmod p = b \bmod p \implies pdifference\ a\ b\ \bmod\ p = 0$ 
  apply auto
  apply (drule mod-eqD [simplified], auto)
  apply (subst mod-div-equality' [of b])
  apply (subst diff-add-assoc2)
  apply (subst mult-commute, subst mult-div-cancel)
  apply simp+
  apply (subst diff-mult-distrib [symmetric])
  apply simp
  apply (drule mod-eqD [simplified], auto)
  apply (subst mod-div-equality' [of b])
  apply (subst diff-add-diff)

```

```

    apply (subst mult-commute, subst mult-div-cancel, auto)
    apply (subst diff-mult-distrib [symmetric])
    apply simp
    done

lemma primekeyrewrite:  $\llbracket \text{prime } p; p \text{ dvd } (a*b); \sim(p \text{ dvd } a) \rrbracket \implies p \text{ dvd } b$ 
    apply (drule prime-dvd-mult)
    apply auto
    done

lemma multzero:  $\llbracket 0 < m \text{ mod } p; m*a = 0 \rrbracket \implies (a::\text{nat}) = 0$ 
    by auto

lemma primekeytrick:  $\llbracket (M*A) \text{ mod } P = (M*B) \text{ mod } P; M \text{ mod } P \neq 0; \text{prime } P \rrbracket$ 
 $\implies A \text{ mod } P = (B::\text{nat}) \text{ mod } P$ 
    apply (drule equalmodstrick2)
    apply (rule equalmodstrick1)
    apply (rule multzero, simp)
    apply (subst mod-mult-distrib2)
    apply (subst timesdistributesoverpdifference)
    apply simp
    apply (rule conjI, rule impI, simp)
    apply (subst diff-mult-distrib2 [symmetric])
    apply (simp add: dvd-eq-mod-eq-0 [symmetric])
    apply (rule primekeyrewrite, simp)
    apply (subst diff-mult-distrib2)
    apply (simp add: dvd-eq-mod-eq-0)+
    apply (rule impI, simp)
    apply (subst diff-mult-distrib2 [symmetric])
    apply (simp add: dvd-eq-mod-eq-0 [symmetric])
    apply (rule disjI2)
    apply (rule primekeyrewrite)
    apply (simp add: dvd-eq-mod-eq-0 diff-mult-distrib2)+
    done

end

```

## 7 Lemmata for modular arithmetic with primes

```

theory Productdivides
imports Pdifference
begin

```

```

lemma productdivides-lemma:  $\llbracket x \text{ mod } z = (0::\text{nat}) \rrbracket \implies ((y*x) \text{ mod } (y*z) = 0)$ 
    apply (subst mod-eq-0-iff [of y*x y*z])
    apply auto
    done

```

```

lemma productdivides:  $\llbracket x \text{ mod } a = 0 :: \text{nat}; x \text{ mod } b = 0; \text{prime } a; \text{prime } b; a \neq b \rrbracket \implies x \text{ mod } (a*b) = 0$ 
  apply (simp add: mod-eq-0-iff [of x a])
  apply (erule exE)
  apply (simp)
  apply (rule productdivides-lemma)
  apply (simp add: dvd-eq-mod-eq-0 [symmetric])
  apply (drule prime-dvd-mult [of b])
  apply (assumption)
  apply (erule disjE)
  apply auto
  apply (simp add: prime-def)
  apply auto
  done

```

```

lemma specializedtoprimes1:  $\llbracket \text{prime } p; \text{prime } q; p \neq q; a \text{ mod } p = b \text{ mod } p; a \text{ mod } q = b \text{ mod } q \rrbracket$ 
   $\implies a \text{ mod } (p*q) = b \text{ mod } (p*q)$ 
  apply (drule equalmodstrick2)+
  apply (rule equalmodstrick1)
  apply (rule productdivides)
  apply auto
  done

```

```

lemma specializedtoprimes1a:
   $\llbracket \text{prime } p; \text{prime } q; p \neq q; a \text{ mod } p = b \text{ mod } p; a \text{ mod } q = b \text{ mod } q; b < p*q \rrbracket$ 
   $\implies a \text{ mod } (p*q) = b$ 
  apply (subst specializedtoprimes1)
  apply auto
  done

```

end

## 8 Pigeon hole principle

```

theory Pigeonholeprinciple
imports Productdivides
begin

```

This theory is a formal proof for the pigeon hole principle. The basic principle is, that if you have to put  $n + 1$  pigeons in  $n$  holes there is at least one hole with more than one pigeon.

```

primrec alldistinct :: nat list  $\Rightarrow$  bool
where
  alldistinct [] = True
| alldistinct (x # xs) = ( $\neg$  x mem xs  $\wedge$  alldistinct xs)

```

**primrec** *alliceseq* :: *nat list*  $\Rightarrow$  *nat*  $\Rightarrow$  *bool*

**where**

*alliceseq* [] = *True*  
| *alliceseq* (*x # xs*) *n* = (*x*  $\leq$  *n*  $\wedge$  *alliceseq xs n*)

**primrec** *allnonzero* :: *nat list*  $\Rightarrow$  *bool*

**where**

*allnonzero* [] = *True*  
| *allnonzero* (*x # xs*) = (*x*  $\neq$  0  $\wedge$  *allnonzero xs*)

**primrec** *positives* :: *nat*  $\Rightarrow$  *nat list*

**where**

*positives* 0 = []  
| *positives* (*Suc n*) = *Suc n # positives n*

**primrec** *timeslist* :: *nat list*  $\Rightarrow$  *nat*

**where**

*timeslist* [] = 1  
| *timeslist* (*x # xs*) = *x \* timeslist xs*

**primrec** *fac* :: *nat*  $\Rightarrow$  *nat*

**where**

*fac* 0 = 1  
| *fac* (*Suc n*) = *Suc n \* fac n*

**primrec** *del* :: *nat*  $\Rightarrow$  *nat list*  $\Rightarrow$  *nat list*

**where**

*del a* [] = []  
| *del a* (*x # xs*) = (if *a*  $\neq$  *x* then *x # del a xs* else *xs*)

**lemma** *length-del*: *x mem xs*  $\implies$  *length (del x xs)* < *length xs*

**by** (*induct xs*) *auto*

**function** *pigeonholeinduction*

**where**

*pigeonholeinduction* [] = *True*  
| *pigeonholeinduction* (*x # xs*) =  
  (if ((*length (x # xs)*) *mem xs*)  
  then (*pigeonholeinduction (del (length (x # xs)) (x # xs))*)  
  else (*pigeonholeinduction xs*))

**by** *pat-completeness auto*

**termination by** (*relation measure size*) (*auto simp add: length-del*)

**lemma** *old-pig-induct*:

**fixes** *P*

**assumes** *P* []

**and** ( $\bigwedge$ (*x::nat*) *xs::nat list*.

$\neg$  *length (x # xs mem xs*  $\longrightarrow$  *P xs*  $\implies$

```

    length (x # xs) mem xs  $\longrightarrow$  P (del (length (x # xs)) (x # xs))  $\implies$ 
      P (x # xs)
  shows P x
  apply (rule pigeonholeinduction.induct)
  using assms apply auto
  done

definition
  perm :: nat list  $\Rightarrow$  nat list  $\Rightarrow$  bool where
    perm xs ys  $\longleftrightarrow$  sort xs = sort ys

lemma allnonzerodelete: allnonzero xs  $\implies$  allnonzero (del x xs)
  by (induct xs) auto

lemma notmemnotdelmem: x  $\neq$  a  $\implies$   $\neg$  a mem xs  $\implies$   $\neg$  a mem (del x xs)
  by (induct xs) auto

lemma alldistinctdelete: alldistinct xs  $\implies$  alldistinct (del x xs)
  apply (induct xs)
  apply auto
  apply (insert notmemnotdelmem)
  apply auto
  done

lemma pigeonholeprinciple-lemma2:  $\neg$  (Suc n) mem xs  $\implies$  allesseq xs (Suc n)
 $\implies$  allesseq xs n
  apply (atomize (full))
  apply (induct xs)
  apply auto
  done

lemma pigeonholeprinciple-lemma1:
  alldistinct xs  $\implies$  allesseq xs (Suc n)  $\implies$  allesseq (del (Suc n) xs) n
  apply (induct xs)
  apply auto
  apply (rule pigeonholeprinciple-lemma2)
  apply auto
  done

lemma anotinsort: a  $\neq$  x  $\implies$  a mem b = a mem (insort x b)
  by (induct b) auto

lemma ainsort: a mem (insort a b)
  by (induct b) auto

lemma memeqsort: x mem xs = x mem (sort xs)
  apply (induct xs)
  apply simp
  apply (case-tac x=a)

```

```

apply (simp add: ainsort)+
apply (rule anotinsort, simp)
done

lemma permmember:  $\llbracket \text{perm } xs \text{ } ys; x \text{ mem } xs \rrbracket \implies x \text{ mem } ys$ 
by (simp add: perm-def memeqsort [of x xs] memeqsort [of x ys])

lemma allessseqdelete:  $\llbracket \text{alldistinct } (x\#xs); \text{allessseq } (x\#xs) (\text{length}(x\#xs)) \rrbracket$ 
 $\implies \text{allessseq } (\text{del } (\text{length}(x\#xs)) (x\#xs)) (\text{length } xs)$ 
apply (insert pigeontholeprinciple-lemma1 [of x#xs length xs])
apply simp
done

lemma perminsert:  $\text{perm } xs \text{ } ys \implies \text{perm } (a\#xs) (a\#ys)$ 
by (simp add: perm-def)

lemma lengthdel2:  $a \text{ mem } xs \implies \text{length}(\text{del } a (x\#xs)) = \text{length } xs$ 
by (induct xs) auto

lemma dellengthinallessseq:
 $\llbracket \text{alldistinct } (x\#xs); \text{allessseq } (x\#xs) (\text{length } (x\#xs)); \text{length } (x\#xs) \text{ mem } xs \rrbracket$ 
 $\implies \text{allessseq } (\text{del } (\text{length } (x\#xs)) (x\#xs))(\text{length } (\text{del } (\text{length } (x\#xs)) (x\#xs)))$ 
apply (drule allessseqdelete)
apply simp
apply (insert lengthdel2 [of length (x#xs) xs x])
apply simp
done

lemma lengthmem:  $\llbracket \text{length } (x\#xs) \text{ mem } (xs) \rrbracket \implies \text{length } (x\#xs) \text{ mem } (x\#xs)$ 
by auto

lemma permSuclengthdel:
 $\llbracket \text{Suc } (\text{length } xs) \text{ mem } xs;$ 
 $\text{perm } (\text{positives } (\text{length } xs)) (x \# \text{del } (\text{Suc } (\text{length } xs)) xs);$ 
 $x \neq \text{Suc } (\text{length } xs) \rrbracket \implies$ 
 $\text{perm } (\text{Suc } (\text{length } xs) \# \text{positives } (\text{length } xs)) ((\text{Suc } (\text{length } xs))\#(x \# \text{del } (\text{Suc}$ 
 $(\text{length } xs)) xs))$ 
apply (rule perminsert)
apply simp
done

lemma insortsym:  $\text{insort } a (\text{insort } x \text{ } xs) = \text{insort } x (\text{insort } a \text{ } xs)$ 
by (induct xs) auto

lemma insortsortdel:  $x \text{ mem } xs \implies \text{insort } x (\text{sort } (\text{del } x \text{ } xs)) = (\text{sort } xs)$ 
apply (induct xs)
apply auto
apply (subst insortsym)
apply simp

```

**done**

**lemma** *permSuclengthdel2*:

[[*Suc* (*length xs*) *mem xs*;  $x \neq \text{Suc } (\text{length } xs)$ ;  
perm (*Suc* (*length xs*) # *positives* (*length xs*)) ((*Suc* (*length xs*))#(*x* # *del* (*Suc*  
(*length xs*) *xs*))]]  
⇒ perm (*Suc* (*length xs*) # *positives* (*length xs*)) (*x* # *xs*)  
**apply** (*simp add: perm-def*)  
**apply** (*subst insortsym*)  
**apply** (*insert insortsortdel [of Suc (length xs) xs, symmetric]*)  
**apply auto**  
**done**

**lemma** *dellengthinperm*:

[[*length* (*x* # *xs*) *mem* (*x*#*xs*);  
perm (*positives* (*length* (*del* (*length* (*x* # *xs*)) (*x* # *xs*))))(*del* (*length* (*x* # *xs*))  
(*x* # *xs*))]]  
⇒ perm (*positives* (*length* (*x* # *xs*))) (*x* # *xs*)  
**apply** (*cases x = Suc (length xs)*)  
**apply simp**  
**apply** (*drule perminsert*)  
**apply simp**  
**apply** (*insert lengthdel2 [of length (x # xs) xs x]*)  
**apply** (*simp del: perm-def positives.simps*)  
**apply** (*simp only: positives.simps*)  
**apply** (*frule permSuclengthdel*)  
**apply simp**  
**apply simp**  
**apply** (*rule permSuclengthdel2*)  
**apply simp-all**  
**done**

**lemma** *positiveseq*: *positives* (*length xs*) = *rev* ([1 ..< *Suc*(*length xs*)])  
**by** (*induct xs*) *auto*

**lemma** *memsetpositives*:

[[perm (*positives* (*length xs*)) *xs*;  $0 < x$ ;  $x \leq \text{length } xs$ ] ⇒  $x \in \text{set } (\text{positives}$   
(*length xs*))  
**apply** (*subst positiveseq*)  
**apply** (*simp add:set-upt*)  
**apply auto**  
**done**

**lemma** *pigeonholeprinciple*:

*allnonzero xs* ⇒ *alldistinct xs* ⇒ *allesseq xs* (*length xs*) ⇒ perm (*positives*  
(*length xs*)) *xs*  
**proof** (*induct xs rule: pigeonholeinduction.induct*)  
**case 1 then show ?case by** (*simp add: perm-def*)  
**next**

```

case (2 x xs) then show ?case
thm notE [of allnonzero (del (length (x # xs)) (x # xs))]
oops

```

**lemmas** seteqmem = mem-iff [symmetric]

**lemma** pigeonholeprinciple:

```

  allnonzero xs  $\implies$  alldistinct xs  $\implies$  allesseq xs (length xs)  $\implies$  perm (positives
(length xs)) xs
  apply (atomize (full))
  apply (induct xs rule: old-pig-induct)
  apply (simp add: perm-def)
  apply safe
  apply (erule-tac P=allnonzero (del (length (x # xs)) (x # xs)) in notE)
  apply (rule allnonzerodelete)
  apply (simp)
  apply (erule-tac P=alldistinct (del (length (x # xs)) (x # xs)) in notE)
  apply (rule alldistinctdelete)
  apply (simp)
  apply (erule-tac P=allesseq (del (length (x # xs)) (x # xs))(length (del (length
(x # xs)) (x # xs))) in notE)
  apply (rule dellengthinallesseq)
  apply (simp)
  apply (simp)
  apply (simp)
  apply (erule-tac P=perm (positives (length (x # xs))) (x # xs) in notE)
  apply (drule lengthmem)
  apply (drule dellengthinperm)
  apply (simp)+
  apply (erule conjE)+
  apply (erule-tac P=allesseq xs (length xs) in notE)
  apply (erule pigeonholeprinciple-lemma2)
  apply (simp)+
  apply (erule conjE)+
  apply (case-tac x=Suc(length xs))
  apply (simp)
  apply (rule perminsert)
  apply (simp)
  apply (drule le-neq-implies-less)
  apply (simp add:less-Suc-eq-le)
  apply (erule-tac P=x mem xs in notE)
  apply (simp add:seteqmem [symmetric])
  apply (frule memsetpositives)
  apply (simp add:seteqmem)+
  apply (rule permmember)
  apply (simp)
  apply (simp)
  apply (simp)
  apply (simp)

```

```

apply (simp)
apply (simp)
apply (simp)
apply (erule-tac P=allnonzero (del (length (x # xs)) (x # xs)) in notE, rule
allnonzerodelete, simp)+
apply (simp)
apply (simp)
apply (simp)
apply (erule-tac P=alldistinct (del (length (x # xs)) (x # xs)) in notE, rule
alldistinctdelete, simp)+
apply (erule-tac P=alllesseq (del (length (x # xs)) (x # xs))(length (del (length
(x # xs)) (x # xs))) in notE)
apply (case-tac length (x#xs) mem xs)
apply (erule dellengthinalllesseq, simp, simp)+
apply (erule-tac P=alllesseq xs (length xs) in notE)
apply (rule pigeonholeprinciple-lemma2)
apply (simp)+
apply (case-tac length (x#xs) mem (x#xs))
apply (frule dellengthinperm)
apply (simp)+
apply (case-tac Suc (length xs) ≠ x)
apply (simp)
apply (erule conjE)+
apply (erule-tac P=alllesseq xs (length xs) in notE)
apply (drule pigeonholeprinciple-lemma2)
apply (simp)
apply (simp)
apply (erule conjE)+
apply (simp)+
apply (case-tac Suc (length xs) = x)
apply (drule perminsert)
apply (simp)+
apply (erule conjE)+
apply (drule le-neq-implies-less)
apply (simp add:less-Suc-eq-le)+
apply (erule-tac P=x mem xs in notE)
apply (simp add:seteqmem [symmetric])
apply (frule memsetpositives)
apply (simp add:seteqmem)+
apply (rule permmember)
apply (simp)+
apply (case-tac Suc (length xs) = x)
apply (drule perminsert)
apply (simp)+
apply (erule conjE)+
apply (drule le-neq-implies-less)
apply (simp add:less-Suc-eq-le)+
apply (erule-tac P=x mem xs in notE)
apply (simp add:seteqmem [symmetric])

```

```

apply (frule memsetpositives)
apply (simp add: seteqmem)+
apply (rule permmember)
apply (simp)+
done

lemma equaltimeslist:  $\llbracket \text{sort } xs = \text{sort } ys \rrbracket \implies \text{timeslist } (\text{sort } xs) = \text{timeslist } (\text{sort } ys)$ 
by auto

lemma timeslistmultkom:  $\text{timeslist } (xs) * x = x * \text{timeslist } (xs)$ 
by simp

lemma timeslistinsort:  $\text{timeslist } (\text{insort } a \ xs) = \text{timeslist } (a\#xs)$ 
by (induct xs) auto

lemma timeslistteq:  $\text{timeslist } (\text{sort } xs) = \text{timeslist } xs$ 
apply (induct xs)
apply auto
apply (simp add: timeslistmultkom [symmetric])
apply (simp add: timeslistinsort)
done

lemma permtimeslist:  $\text{perm } xs \ ys \implies \text{timeslist } xs = \text{timeslist } ys$ 
apply (simp only: perm-def)
apply (insert equaltimeslist [of xs ys])
apply (simplesubst timeslistteq [symmetric])
apply (simplesubst timeslistteq [of xs, symmetric])
apply auto
done

lemma timeslistpositives:  $\text{timeslist } (\text{positives } n) = \text{fac } n$ 
by (induct n) auto

lemma pdvdnot:  $\llbracket \text{prime } p; \neg p \text{ dvd } x; \neg p \text{ dvd } y \rrbracket \implies \neg p \text{ dvd } x*y$ 
apply (auto)
apply (insert prime-dvd-mult [of p x y])
apply simp
done

lemma lessdvdnot:  $\llbracket \text{Suc } (x::\text{nat}) < p \rrbracket \implies \neg p \text{ dvd } \text{Suc } x$ 
apply auto
apply (frule mod-less)
apply (frule dvd-imp-le)
apply auto
done

lemma pnotdvdall:  $\llbracket \text{prime } p; p \text{ dvd } (\text{Suc } n) * (\text{fac } n); \neg p \text{ dvd } \text{fac } n; \text{Suc } n < p \rrbracket \implies \text{False}$ 

```

```

apply (insert lessdvdnot [of n p])
apply (insert pdvdnot [of p Suc n fac n])
apply auto
done

lemma primefact: prime p  $\implies$  (n::nat) < p  $\implies$  fac n mod p  $\neq$  0
apply (induct n)
  apply (simp add: prime-def)
apply (simp only: fac.simps dvd-eq-mod-eq-0 [symmetric])
apply clarify
apply (drule meta-mp)
  apply simp
apply (insert lessdvdnot pdvdnot [of p fac n Suc n] pnotdvdall)
apply auto
done

end

```

## 9 Fermats little theorem

```

theory Fermat
imports Pigeonholeprinciple
begin

primrec pred:: nat  $\Rightarrow$  nat
where
  pred 0 = 0
| pred (Suc a) = a

primrec S :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat list
where
  S 0 M P = []
| S (Suc N) M P = (M * Suc N) mod P # S N M P

lemma remaindertimeslist: timeslist (S n M p) mod p = fac n * M ^ n mod p
proof (induct n)
  case 0 then show ?case by simp
next
  case (Suc n) then show ?case
    apply auto
    apply (simp add: add-mult-distrib)
    apply (simp add: mult-assoc [symmetric])
    apply (subst add-mult-distrib [symmetric])
    apply (subst mult-assoc)
    apply (subst mult-left-commute)
    apply (subst add-mult-distrib2 [symmetric])
    apply (simp add: mult-assoc)
    apply (subst mult-left-commute)

```

```

    apply (simp add: mult-commute [of M])
    apply (subst mod-mult-left-eq [of M + n * M])
    apply (erule remainderexplemma)
  done
qed

lemma succassoc: (P + P*w) = P * Suc w
  by auto

lemma modI: 0 < (x::nat) mod p  $\implies$  0 < x
  by (induct x) auto

lemma delmulmod:  $\llbracket 0 < x \text{ mod } p; a < (b::\text{nat}) \rrbracket \implies x*a < x*b$ 
  apply (simp, rule modI, simp)
  done

lemma swaple: (c < b)  $\implies$  ((a::nat)  $\leq$  b - c)  $\implies$  c  $\leq$  b - a
  by arith

lemma exchgmin:  $\llbracket (a::\text{nat}) < b; c \leq a-b \rrbracket \implies c \leq a - a$ 
  by auto

lemma sucleI: Suc x  $\leq$  0  $\implies$  False
  by auto

lemma diffI: (0::nat) = b - b
  by auto

lemma alldistincts: prime p  $\implies$  (m mod p  $\neq$  0)  $\implies$  (n2 < n1)  $\implies$  (n1 < p)
 $\implies$ 
   $\neg(((m*n1) \text{ mod } p) \text{ mem } (S \ n2 \ m \ p))$ 
  apply (induct n2)
  apply auto
  apply (drule equalmodstrick2)
  apply (subgoal-tac m+m*n2 < m*n1)
  apply auto
  apply (drule dvdI)
  apply (simp only: succassoc diff-mult-distrib2[symmetric])
  apply (drule primekeyrewrite, simp)
  apply (simp add: dvd-eq-mod-eq-0)
  apply (drule-tac n = n1 - Suc n2 in dvd-imp-le, simp)
  apply (rule sucleI, subst diffI [of n1])
  apply (rule exchgmin, simp)
  apply (rule swaple, auto)
  apply (subst succassoc)
  apply (rule delmulmod)
  apply auto
  done

```

```

lemma alldistincts2: prime p  $\implies$  m mod p  $\neq$  0  $\implies$  n < p  $\implies$  alldistinct (S n
m p)
  apply (induct n)
  apply (simp)+
  apply (subst succassoc)
  apply (rule alldistincts)
  apply auto
  done

lemma notdvdless:  $\neg$  a dvd b  $\implies$  0 < (b::nat) mod a
  apply (rule contrapos-np, simp)
  apply (simp add: dvd-eq-mod-eq-0)
  done

lemma allnonzerop: prime p  $\implies$  m mod p  $\neq$  0  $\implies$  n < p  $\implies$  allnonzero (S n m
p)
  apply (induct n)
  apply simp+
  apply (subst succassoc)
  apply (rule notdvdless)
  apply clarify
  apply (drule primekeyrewrite)
  apply assumption
  apply (simp add: dvd-eq-mod-eq-0)
  apply (drule-tac n = Suc n in dvd-imp-le)
  apply auto
  done

lemma predI: a < p  $\implies$  a  $\leq$  pred p
  by (induct p) auto

lemma predd: pred p = p - (1::nat)
  by (induct p) auto

lemma allessseqps: p  $\neq$  0  $\implies$  allessseq (S n m p) (pred p)
  by (induct n) (auto simp add: predI mod-less-divisor)

lemma lengths: length (S n m p) = n
  by (induct n) auto

lemma suconeless: prime p  $\implies$  p - 1 < p
  by (induct p) (auto simp add: prime-def)

lemma primenotzero: prime p  $\implies$  p  $\neq$  0
  by (auto simp add: prime-def)

lemma onemodprime: prime p  $\implies$  1 mod p = (1::nat)
  by (induct p) (auto simp add: prime-def)

```

```

lemma fermat:  $\llbracket \text{prime } p; m \bmod p \neq 0 \rrbracket \implies m^{(p-(1::\text{nat}))} \bmod p = 1$ 
  apply (frule onemodprime[symmetric], simp)
  apply (frule-tac  $n = p - \text{Suc } 0$  in primefact)
  apply (drule suconeless, simp)
  apply (erule ssubst)
  back
  apply (rule-tac  $M = \text{fac } (p - \text{Suc } 0)$  in primekeytrick)
  apply (subst remaindertimeslist [of  $p - \text{Suc } 0$   $m$   $p$ , symmetric])
  apply (frule-tac  $n = p - (1::\text{nat})$  in alldistincts2, simp)
  apply (rule suconeless, simp)
  apply (frule-tac  $n = p - (1::\text{nat})$  in allnonzerop, simp)
  apply (rule suconeless, simp)
  apply (frule primenotzero)
  apply (frule-tac  $n = p - (1::\text{nat})$  and  $m = m$  and  $p = p$  in allesseqps)
  apply (frule primenotzero)
  apply (simp add: predd)
  apply (insert lengths[of  $p - \text{Suc } 0$   $m$   $p$ , symmetric])
  apply (insert pigeonholeprinciple [of  $S$   $(p - \text{Suc } 0)$   $m$   $p$ ])
  apply (auto)
  apply (drule permtimeslist)
  apply (simp add: timeslistpositives)
  done

end

```

## 10 Correctness proof for RSA

```

theory Cryptinverts
imports Fermat Crypt
begin

```

In this theory we show, that a RSA encrypted message can be decrypted

```

lemma cryptinverts-hilf1:  $\text{prime } p \implies (m * m^{(k * \text{pred } p)}) \bmod p = m \bmod p$ 
  apply (cases  $m \bmod p = 0$ )
  apply (simp add: mod-mult-left-eq)
  apply (simp only: mult-commute [of  $k$   $\text{pred } p$ ]
    power-mult mod-mult-right-eq [of  $m$   $(m^{\text{pred } p})^k$   $p$ ]
    remainderexp [of  $m^{\text{pred } p}$   $p$   $k$ , symmetric])
  apply (insert fermat [of  $p$   $m$ ])
  apply (simp add: predd)
  apply (subst One-nat-def [symmetric])
  apply (subst onemodprime)
  apply auto
  done

```

```

lemma cryptinverts-hilf2:  $\text{prime } p \implies m * (m^{(k * (\text{pred } p) * (\text{pred } q))}) \bmod p = m \bmod p$ 
  apply (simp add: mult-commute [of  $k * \text{pred } p$   $\text{pred } q$ ] mult-assoc [symmetric])

```

```

apply (rule cryptinverts-hilf1 [of p m (pred q) * k])
apply simp
done

lemma cryptinverts-hilf3: prime q  $\implies m*(m^(k * (pred p) * (pred q))) \bmod q = m \bmod q$ 
apply (simp only: mult-assoc)
apply (simp add: mult-commute [of pred p pred q])
apply (simp only: mult-assoc [symmetric])
apply (rule cryptinverts-hilf2)
apply simp
done

lemma cryptinverts-hilf4:
   $\llbracket \text{prime } p; \text{prime } q; p \neq q; m < p*q; x \bmod ((\text{pred } p)*(\text{pred } q)) = 1 \rrbracket \implies m^x \bmod (p*q) = m$ 
apply (frule cryptinverts-hilf2 [of p m k q])
apply (frule cryptinverts-hilf3 [of q m k p])
apply (frule mod-eqD)
apply (elim exE)
apply (rule specializedtoprimes1a)
apply (simp add: cryptinverts-hilf2 cryptinverts-hilf3 mult-assoc [symmetric])
done

lemma primmultgreater:  $\llbracket \text{prime } p; \text{prime } q; p \neq 2; q \neq 2 \rrbracket \implies 2 < p*q$ 
apply (simp add: prime-def)
apply (insert mult-le-mono [of 2 p 2 q])
apply auto
done

lemma primmultgreater2:  $\llbracket \text{prime } p; \text{prime } q; p \neq q \rrbracket \implies 2 < p*q$ 
apply (cases p = 2)
apply simp+
apply (simp add: prime-def)
apply (cases q = 2)
apply (simp add: prime-def)
apply (erule primmultgreater)
apply auto
done

lemma cryptinverts:  $\llbracket \text{prime } p; \text{prime } q; p \neq q; n = p*q; m < n; e*d \bmod ((\text{pred } p)*(\text{pred } q)) = 1 \rrbracket \implies \text{rsa-crypt } (\text{rsa-crypt } m e n) d n = m$ 
apply (insert cryptinverts-hilf4 [of p q m e*d])
apply (insert cryptcorrect [of p*q rsa-crypt m e (p * q) d])
apply (insert cryptcorrect [of p*q m e])
apply (insert primmultgreater2 [of p q])
apply (simp add: prime-def)
apply (simp add: remainderexp [of m^e p*q d] power-mult [symmetric])
done

```

end

## 11 Extensions to the Word theory required for PSS

**theory** Wordarith

**imports** WordOperations  $\sim\sim$  /src/HOL/Old-Number-Theory/Primes

**begin**

**definition**

*nat-to-bv-length* ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{bv}$  **where**

*nat-to-bv-length*:

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

**lemma** *length-nat-to-bv-length*:

*nat-to-bv-length*  $x\ y \neq [] \implies \text{length} (\text{nat-to-bv-length } x\ y) = y$

**unfolding** *nat-to-bv-length* **by** *auto*

**lemma** *bv-to-nat-nat-to-bv-length*:

*nat-to-bv-length*  $x\ y \neq [] \implies \text{bv-to-nat} (\text{nat-to-bv-length } x\ y) = x$

**unfolding** *nat-to-bv-length* **by** *auto*

**definition**

*roundup* ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$  **where**

*roundup*: *roundup*  $x\ y = (\text{if } (x \bmod y = 0) \text{ then } (x \text{ div } y) \text{ else } (x \text{ div } y) + 1)$

**lemma** *rndvdvd*:  $b \text{ dvd } a \implies \text{roundup } a\ b * b = a$

**by** (*auto simp add: roundup dvd-eq-mod-eq-0*)

**lemma** *bv-to-nat-zero-prepend*:  $\text{bv-to-nat } a = \text{bv-to-nat } (\mathbf{0}\#a)$

**by** *auto*

**primrec** *remzero*::  $\text{bv} \Rightarrow \text{bv}$  **where**

*remzero*  $[] = []$

| *remzero*  $(a\#b) = (\text{if } a = \mathbf{1} \text{ then } (a\#b) \text{ else } \text{remzero } b)$

**lemma** *remzeroeq*:  $\text{bv-to-nat } a = \text{bv-to-nat} (\text{remzero } a)$

**proof** (*induct a*)

**show**  $\text{bv-to-nat } [] = \text{bv-to-nat} (\text{remzero } [])$  **by** (*simp add: remzero.simps*)

**next**

**case** (*Cons a1 a2*)

```

show  $bv\text{-to-nat } a2 = bv\text{-to-nat } (remzero a2) \implies$ 
   $bv\text{-to-nat } (a1 \# a2) = bv\text{-to-nat } (remzero (a1 \# a2))$ 
proof (cases a1)
  assume  $a: a1=0$  then have  $bv\text{-to-nat } (a1 \# a2) = bv\text{-to-nat } a2$ 
    using bv-to-nat-zero-prepend by simp
  moreover have  $remzero (a1 \# a2) = remzero a2$  using  $a$  by simp
  ultimately show ?thesis using Cons by simp
next
  assume  $a1=1$  then show ?thesis by simp
qed
qed

```

```

lemma len-nat-to-bv-pos: assumes  $x: 1 < a$  shows  $0 < length (nat\text{-to-bv } a)$ 
proof (auto)
  assume  $b: nat\text{-to-bv } a = []$ 
  have  $a: bv\text{-to-nat } [] = 0$  by simp
  have  $c: bv\text{-to-nat } (nat\text{-to-bv } a) = 0$  using  $a$  and  $b$  by simp
  from  $x$  have  $d: bv\text{-to-nat } (nat\text{-to-bv } a) = a$  by simp
  from  $d$  and  $c$  have  $a=0$  by simp
  then show False using  $x$  by simp
qed

```

```

lemma remzero-replicate:  $remzero ((replicate n 0)@l) = remzero l$ 
by (induct n, auto)

```

```

lemma length-bvxor-bound:  $a \leq length l \implies a \leq length (bv\text{xor } l l2)$ 
proof (induct a)
  case 0
  then show ?case by simp
next
  case (Suc a)
  have  $a: Suc a \leq length l$  by fact
  with Suc.hyps have  $b: a \leq length (bv\text{xor } l l2)$  by simp
  show  $Suc a \leq length (bv\text{xor } l l2)$ 
  proof cases
    assume  $c: a = length (bv\text{xor } l l2)$ 
    show  $Suc a \leq length (bv\text{xor } l l2)$ 
    proof (simp add: vxor)
      have  $length l \leq max (length l) (length l2)$  by simp
      then show  $Suc a \leq max (length l) (length l2)$  using  $a$  by simp
    qed
  next
    assume  $a \neq length (bv\text{xor } l l2)$ 
    then have  $a < length (bv\text{xor } l l2)$  using  $b$  by simp
    then show ?thesis by simp
  qed
qed

```

```

lemma len-lower-bound:

```

```

assumes  $0 < n$ 
shows  $2^{\wedge}(\text{length } (\text{nat-to-bv } n) - \text{Suc } 0) \leq n$ 
proof (cases  $1 < n$ )
  assume  $l1: 1 < n$ 
  then show ?thesis
  proof (simp add: nat-to-bv-def, induct n rule: nat-to-bv-helper.induct, auto)
    fix  $n$ 
    assume  $a: \text{Suc } 0 < (n::\text{nat})$  and  $b: \sim \text{Suc } 0 < n \text{ div } 2$ 
    then have  $n=2 \vee n=3$ 
    proof (cases  $n \leq 3$ )
      assume  $n \leq 3$  and  $\text{Suc } 0 < n$ 
      then show  $n=2 \vee n=3$  by auto
    next
      assume  $\sim n \leq 3$  then have  $3 < n$  by simp
      then have  $1 < n \text{ div } 2$  by arith
      then show  $n=2 \vee n=3$  using  $b$  by simp
    qed
    then show  $2^{\wedge}(\text{length } (\text{nat-to-bv-helper } n []) - \text{Suc } 0) \leq n$ 
    proof (cases  $n=2$ )
      assume  $a: n=2$  then have  $\text{nat-to-bv-helper } n [] = [1, 0]$ 
      proof –
        have  $\text{nat-to-bv-helper } n [] = \text{nat-to-bv } n$  using  $b$  by (simp add: nat-to-bv-def)
        then show ?thesis using  $a$  by (simp add: nat-to-bv-non0)
      qed
      then show  $2^{\wedge}(\text{length } (\text{nat-to-bv-helper } n []) - \text{Suc } 0) \leq n$  using  $a$  by simp
    next
      assume  $n=2 \vee n=3$  and  $n \sim 2$ 
      then have  $a: n=3$  by simp
      then have  $\text{nat-to-bv-helper } n [] = [1, 1]$ 
      proof –
        have  $\text{nat-to-bv-helper } n [] = \text{nat-to-bv } n$  using  $a$  by (simp add: nat-to-bv-def)
        then show ?thesis using  $a$  by (simp add: nat-to-bv-non0)
      qed
      then show  $2^{\wedge}(\text{length } (\text{nat-to-bv-helper } n []) - \text{Suc } 0) \leq n$  using  $a$  by simp
    qed
  next
    fix  $n$ 
    assume  $a: \text{Suc } 0 < n$  and
       $b: 2^{\wedge}(\text{length } (\text{nat-to-bv-helper } (n \text{ div } 2) []) - \text{Suc } 0) \leq n \text{ div } 2$ 
    have  $(2::\text{nat})^{\wedge}(\text{length } (\text{nat-to-bv-helper } n []) - \text{Suc } 0) =$ 
       $2^{\wedge}(\text{length } (\text{nat-to-bv-helper } (n \text{ div } 2) []) + 1 - \text{Suc } 0)$ 
    proof –
      have  $\text{length } (\text{nat-to-bv } n) = \text{length } (\text{nat-to-bv } (n \text{ div } 2)) + 1$ 
      using  $a$  by (simp add: nat-to-bv-non0)
      then show ?thesis by (simp add: nat-to-bv-def)
    qed
    moreover have  $(2::\text{nat})^{\wedge}(\text{length } (\text{nat-to-bv-helper } (n \text{ div } 2) []) + 1 - \text{Suc } 0)$ 
  =
     $2^{\wedge}(\text{length } (\text{nat-to-bv-helper } (n \text{ div } 2) []) - \text{Suc } 0) * 2$ 

```

```

proof auto
  have (2::nat)^(length (nat-to-bv-helper (n div 2) [])) - Suc 0 * 2 =
    2^(length (nat-to-bv-helper (n div 2) [])) - Suc 0 + 1 by simp
  moreover have (2::nat)^(length (nat-to-bv-helper (n div 2) [])) - Suc 0 +
1) =
  2^(length (nat-to-bv-helper (n div 2) []))
  proof -
    have 0 < n div 2 using a by arith
    then have 0 < length (nat-to-bv (n div 2)) by (simp add: nat-to-bv-non0)
    then have 0 < length (nat-to-bv-helper (n div 2) []) using a by (simp add:
nat-to-bv-def)
    then show ?thesis by simp
  qed
  ultimately show (2::nat) ^ length (nat-to-bv-helper (n div 2) []) =
    2 ^ (length (nat-to-bv-helper (n div 2) [])) - Suc 0 * 2 by simp
  qed
  ultimately show 2 ^ (length (nat-to-bv-helper n []) - Suc 0) <= n
  using b by (simp add: nat-to-bv-def) arith
  qed
next
  assume c: ~ 1 < n
  show ?thesis
  proof (cases n=1)
    assume a: n=1 then have nat-to-bv n = [1] by (simp add: nat-to-bv-non0)
    then show 2^(length (nat-to-bv n) - Suc 0) <= n using a by simp
  next
    assume n ~ 1
    with (0 < n) show 2^(length (nat-to-bv n) - Suc 0) <= n using c by simp
  qed
qed

lemma length-lower: assumes a: length a < length b and b: (hd b) ~ 0 shows
bv-to-nat a < bv-to-nat b
proof -
  have ha: bv-to-nat a < 2^length a by (simp add: bv-to-nat-upper-range)
  have b ~ [] using a by auto
  then have b=(hd b)#(tl b) by simp
  then have bv-to-nat b = bitval (hd b) * 2^(length (tl b)) + bv-to-nat (tl b) using
bv-to-nat-helper[of hd b tl b] by simp
  moreover have bitval (hd b) = 1
  proof (cases hd b)
    assume hd b = 0
    then show bitval (hd b) = 1 using b by simp
  next
    assume hd b = 1
    then show bitval (hd b) = 1 by simp
  qed
  ultimately have hb: 2^length (tl b) <= bv-to-nat b by simp
  have 2^(length a) <= (2::nat) ^ length (tl b) using a by auto

```

then show *?thesis* using *hb* and *ha* by *arith*  
qed

lemma *nat-to-bv-non-empty*: assumes *a*:  $0 < n$  shows  $\text{nat-to-bv } n \sim = []$   
proof –  
from *nat-to-bv-non0*[of *n*] have  $\exists x. \text{nat-to-bv } n = x @ [\text{if } n \bmod 2 = 0 \text{ then } \mathbf{0} \text{ else } \mathbf{1}]$  using *a* by *simp*  
then show *?thesis* by *auto*  
qed

lemma *hd-append*:  $x \sim = [] \implies \text{hd } (x @ xs) = \text{hd } x$   
by (*induct x*) *auto*

lemma *hd-one*:  $0 < n \implies \text{hd } (\text{nat-to-bv-helper } n []) = \mathbf{1}$   
proof (*induct n rule: nat-to-bv-helper.induct*)

fix *n*  
assume \*:  $n \neq 0 \implies 0 < n \text{ div } 2 \implies \text{hd } (\text{nat-to-bv-helper } (n \text{ div } 2) []) = \mathbf{1}$   
and  $0 < n$   
show  $\text{hd } (\text{nat-to-bv-helper } n []) = \mathbf{1}$   
proof (*cases 1 < n*)  
assume *a*:  $1 < n$   
with \* have *b*:  $0 < n \text{ div } 2 \implies \text{hd } (\text{nat-to-bv-helper } (n \text{ div } 2) []) = \mathbf{1}$  by *simp*  
from *a* have *c*:  $0 < n \text{ div } 2$  by *arith*  
then have *d*:  $\text{hd } (\text{nat-to-bv-helper } (n \text{ div } 2) []) = \mathbf{1}$  using *b* by *simp*  
also from *a* have  $0 < n$  by *simp*  
then have  $\text{hd } (\text{nat-to-bv-helper } n []) =$   
 $\text{hd } (\text{nat-to-bv } (n \text{ div } 2) @ [\text{if } n \bmod 2 = 0 \text{ then } \mathbf{0} \text{ else } \mathbf{1}])$   
using *nat-to-bv-def* and *nat-to-bv-non0*[of *n*] by *auto*  
then have  $\text{hd } (\text{nat-to-bv-helper } n []) =$   
 $\text{hd } (\text{nat-to-bv } (n \text{ div } 2))$   
using *nat-to-bv-non0*[of *n div 2*] and *c* and  
*nat-to-bv-non-empty*[of *n div 2*] and *hd-append*[of *nat-to-bv (n div 2)*] by  
*auto*  
then have  $\text{hd } (\text{nat-to-bv-helper } n []) = \text{hd } (\text{nat-to-bv-helper } (n \text{ div } 2) [])$   
using *nat-to-bv-def* by *simp*  
then show  $\text{hd } (\text{nat-to-bv-helper } n []) = \mathbf{1}$  using *b* and *c* by *simp*  
next  
assume  $\sim 1 < n$  with  $\langle 0 < n \rangle$  have *c*:  $n = 1$  by *simp*  
have  $\text{nat-to-bv-helper } 1 [] = [\mathbf{1}]$  by (*simp add: nat-to-bv-helper.simps*)  
then show  $\text{hd } (\text{nat-to-bv-helper } n []) = \mathbf{1}$  using *c* by *simp*  
qed  
qed

lemma *prime-hd-non-zero*: assumes *a*: *prime p* and *b*: *prime q* shows  $\text{hd } (\text{nat-to-bv } (p * q)) \sim = \mathbf{0}$   
proof –  
have *c*:  $\bigwedge p. \text{prime } p \implies (1 :: \text{nat}) < p$   
proof –  
fix *p*

assume  $d$ : prime  $p$   
 then show  $1 < p$  by (simp add: prime-def)  
 qed  
 have  $1 < p$  using  $c$  and  $a$  by simp  
 moreover have  $1 < q$  using  $c$  and  $b$  by simp  
 ultimately have  $0 < p*q$  by simp  
 then show ?thesis using hd-one[of  $p*q$ ] and nat-to-bv-def by auto  
 qed

lemma primerew:  $\llbracket m \text{ dvd } p; m \sim 1; m \sim p \rrbracket \implies \sim \text{prime } p$   
 by (auto simp add: prime-def)

lemma two-dvd-exp:  $0 < x \implies (2::\text{nat}) \text{ dvd } 2^x$   
 by (induct  $x$ ) auto

lemma exp-prod1:  $\llbracket 1 < b; 2^x = 2*(b::\text{nat}) \rrbracket \implies 2 \text{ dvd } b$

proof –  
 assume  $a$ :  $1 < b$  and  $b$ :  $2^x = 2*(b::\text{nat})$   
 have  $s1$ :  $1 < x$   
 proof (cases  $1 < x$ )  
 assume  $1 < x$  then show ?thesis by simp  
 next  
 assume  $x \sim 1 < x$  then have  $2^x \leq (2::\text{nat})$  using  $b$   
 proof (cases  $x = 0$ )  
 assume  $x = 0$  then show  $2^x \leq (2::\text{nat})$  by simp  
 next  
 assume  $x \sim 0$  then have  $x = 1$  using  $x$  by simp  
 then show  $2^x \leq (2::\text{nat})$  by simp  
 qed  
 then have  $b \leq 1$  using  $b$  by simp  
 then show ?thesis using  $a$  by simp  
 qed  
 have  $s2$ :  $2^{(x-(1::\text{nat}))} = b$   
 proof –  
 from  $s1$   $b$  have  $2^{((x-\text{Suc } 0)+1)} = 2*b$  by simp  
 then have  $2*2^{(x-\text{Suc } 0)} = 2*b$  by simp  
 then show  $2^{(x-(1::\text{nat}))} = b$  by simp  
 qed  
 from  $s1$  and  $s2$  show ?thesis using two-dvd-exp[of  $x-(1::\text{nat})$ ] by simp  
 qed

lemma exp-prod2:  $\llbracket 1 < a; 2^x = a*2 \rrbracket \implies (2::\text{nat}) \text{ dvd } a$

proof –  
 assume  $2^x = a*2$   
 then have  $2^x = 2*a$  by simp  
 moreover assume  $1 < a$   
 ultimately show  $2 \text{ dvd } a$  using exp-prod1 by simp  
 qed

```

lemma odd-mul-odd: [[ $\sim(2::nat)$  dvd p;  $\sim 2$  dvd q]]  $\implies \sim 2$  dvd p*q
  apply (simp add: dvd-eq-mod-eq-0)
  apply (simp add: mod-mult-right-eq)
  done

lemma prime-equal: [[prime p; prime q;  $2^x=p*q$ ]]  $\implies (p=q)$ 
proof -
  assume a: prime p and b: prime q and c:  $2^x=p*q$ 
  from a have d:  $1 < p$  by (simp add: prime-def)
  moreover from b have e:  $1 < q$  by (simp add: prime-def)
  show p=q
  proof (cases p=2)
    assume p: p=2 then have 2 dvd q using c and exp-prod1[of q x] and e by
    simp
    then have 2=q using primerev[of 2 q] and b by auto
    then show ?thesis using p by simp
  next
    assume p: p $\sim$ =2 show p=q
    proof (cases q=2)
      assume q: q=2 then have 2 dvd p using c and exp-prod1[of p x] and d
      by simp
      then have 2=p using primerev[of 2 p] and a by auto
      then show ?thesis using p by simp
    next
      assume q: q $\sim$ =2 show p=q
      proof -
        from p have  $\sim 2$  dvd p using primerev and a by auto
        moreover from q have  $\sim 2$  dvd q using primerev and b by auto
        ultimately have  $\sim 2$  dvd p*q by (simp add: odd-mul-odd)
        moreover have (2::nat) dvd  $2^x$ 
        proof (cases x=0)
          assume x=0 then have (2::nat) $^x$ =1 by simp
          then show ?thesis using c and d and e by simp
        next
          assume x $\sim$ =0 then have 0<x by simp
          then show ?thesis using two-dvd-exp by simp
        qed
        ultimately have  $2^x \sim = p*q$  by auto
        then show ?thesis using c by simp
      qed
    qed
  qed
qed
qed

lemma nat-to-bv-length-bv-to-nat:
  length xs = n  $\implies$  xs  $\neq$  []  $\implies$  nat-to-bv-length (bv-to-nat xs) n = xs
  apply (simp only: nat-to-bv-length)
  apply (auto)

```

```

apply (simp add: bv-extend-norm-unsigned)
done

```

```

end

```

## 12 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

```

consts BC:: bv
        salt:: bv
        sLen:: nat
        generate-M':: bv  $\Rightarrow$  bv  $\Rightarrow$  bv
        generate-PS:: nat  $\Rightarrow$  nat  $\Rightarrow$  bv
        generate-DB:: bv  $\Rightarrow$  bv
        generate-H:: bv  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bv
        generate-maskedDB:: bv  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bv
        generate-salt:: bv  $\Rightarrow$  bv
        show-rightmost-bits:: bv  $\Rightarrow$  nat  $\Rightarrow$  bv
        MGF:: bv  $\Rightarrow$  nat  $\Rightarrow$  bv
        MGF1:: bv  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bv
        MGF2:: bv  $\Rightarrow$  nat  $\Rightarrow$  bv
        maskedDB-zero:: bv  $\Rightarrow$  nat  $\Rightarrow$  bv
        emsapss-encode:: bv  $\Rightarrow$  nat  $\Rightarrow$  bv
        emsapss-encode-help1:: bv  $\Rightarrow$  nat  $\Rightarrow$  bv
        emsapss-encode-help2:: bv  $\Rightarrow$  nat  $\Rightarrow$  bv
        emsapss-encode-help3:: bv  $\Rightarrow$  nat  $\Rightarrow$  bv
        emsapss-encode-help4:: bv  $\Rightarrow$  bv  $\Rightarrow$  nat  $\Rightarrow$  bv
        emsapss-encode-help5:: bv  $\Rightarrow$  bv  $\Rightarrow$  nat  $\Rightarrow$  bv
        emsapss-encode-help6:: bv  $\Rightarrow$  bv  $\Rightarrow$  bv  $\Rightarrow$  nat  $\Rightarrow$  bv
        emsapss-encode-help7:: bv  $\Rightarrow$  bv  $\Rightarrow$  nat  $\Rightarrow$  bv
        emsapss-encode-help8:: bv  $\Rightarrow$  bv  $\Rightarrow$  bv
        emsapss-decode:: bv  $\Rightarrow$  bv  $\Rightarrow$  nat  $\Rightarrow$  bool
        emsapss-decode-help1:: bv  $\Rightarrow$  bv  $\Rightarrow$  nat  $\Rightarrow$  bool
        emsapss-decode-help2:: bv  $\Rightarrow$  bv  $\Rightarrow$  nat  $\Rightarrow$  bool
        emsapss-decode-help3:: bv  $\Rightarrow$  bv  $\Rightarrow$  nat  $\Rightarrow$  bool
        emsapss-decode-help4:: bv  $\Rightarrow$  bv  $\Rightarrow$  bv  $\Rightarrow$  nat  $\Rightarrow$  bool
        emsapss-decode-help5:: bv  $\Rightarrow$  bv  $\Rightarrow$  bv  $\Rightarrow$  bv  $\Rightarrow$  nat  $\Rightarrow$  bool
        emsapss-decode-help6:: bv  $\Rightarrow$  bv  $\Rightarrow$  bv  $\Rightarrow$  nat  $\Rightarrow$  bool
        emsapss-decode-help7:: bv  $\Rightarrow$  bv  $\Rightarrow$  bv  $\Rightarrow$  nat  $\Rightarrow$  bool
        emsapss-decode-help8:: bv  $\Rightarrow$  bv  $\Rightarrow$  bv  $\Rightarrow$  bool
        emsapss-decode-help9:: bv  $\Rightarrow$  bv  $\Rightarrow$  bv  $\Rightarrow$  bool
        emsapss-decode-help10:: bv  $\Rightarrow$  bv  $\Rightarrow$  bool
        emsapss-decode-help11:: bv  $\Rightarrow$  bv  $\Rightarrow$  bool

```

## defs

*show-rightmost-bits:*

*show-rightmost-bits bvec n* == *rev*(*take n* (*rev bvec*) )

*BC:*

*BC* == [*One*, *Zero*, *One*, *One*, *One*, *One*, *Zero*, *Zero*]

*salt:*

*salt* == []

*sLen:*

*sLen* == *length salt*

*generate-M':*

*generate-M' mHash salt-new* == (*bv-prepend 64 0 []*) @ *mHash* @ *salt-new*

*generate-PS:*

*generate-PS emBits hLen* == *bv-prepend* ((*roundup emBits 8*)\*8 - *sLen* - *hLen* - 16) **0 []**

*generate-DB:*

*generate-DB PS* == *PS* @ [*Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *One*] @ *salt*

*maskedDB-zero:*

*maskedDB-zero maskedDB emBits* == *bv-prepend* ((*roundup emBits 8*) \* 8 - *emBits*) **0** (*drop* ((*roundup emBits 8*)\*8 - *emBits*) *maskedDB*)

*generate-H:*

*generate-H EM emBits hLen* == *take hLen* (*drop* ((*roundup emBits 8*)\*8 - *hLen* - 8) *EM*)

*generate-maskedDB:*

*generate-maskedDB EM emBits hLen* == *take* ((*roundup emBits 8*)\*8 - *hLen* - 8) *EM*

*generate-salt:*

*generate-salt DB-zero* == *show-rightmost-bits DB-zero sLen*

*MGF:*

*MGF Z l* == *if* *l* = 0 ∨ 2<sup>32</sup>\*(*length* (*sha1 Z*)) < *l*  
    *then* []  
    *else* *MGF1 Z* ( *roundup l* (*length* (*sha1 Z*)) - 1 ) *l*

*MGF1:*

*MGF1 Z n l* == *take l* (*MGF2 Z n*)

```

emsapss-encode:
emsapss-encode M emBits == if (2^64 ≤ length M ∨ 2^32 * 160 < emBits)
    then []
    else emsapss-encode-help1 (sha1 M) emBits

emsapss-encode-help1:
emsapss-encode-help1 mHash emBits == if emBits < length (mHash) + sLen
+ 16
    then []
    else emsapss-encode-help2 (generate-M' mHash
salt) emBits
emsapss-encode-help2:
emsapss-encode-help2 M' emBits == emsapss-encode-help3 (sha1 M') emBits

emsapss-encode-help3:
emsapss-encode-help3 H emBits == emsapss-encode-help4 (generate-PS emBits
(length H)) H emBits

emsapss-encode-help4:
emsapss-encode-help4 PS H emBits == emsapss-encode-help5 (generate-DB PS)
H emBits

emsapss-encode-help5:
emsapss-encode-help5 DB H emBits == emsapss-encode-help6 DB (MGF H
(length DB)) H emBits

emsapss-encode-help6:
emsapss-encode-help6 DB dbMask H emBits == if dbMask = []
    then []
    else emsapss-encode-help7 (bvxor DB dbMask)
H emBits

emsapss-encode-help7:
emsapss-encode-help7 maskedDB H emBits == emsapss-encode-help8 (maskedDB-zero
maskedDB emBits) H

emsapss-encode-help8:
emsapss-encode-help8 DBzero H == DBzero @ H @ BC

emsapss-decode:
emsapss-decode M EM emBits == if (2^64 ≤ length M ∨ 2^32*160 < emBits)
    then False
    else emsapss-decode-help1 (sha1 M) EM emBits

emsapss-decode-help1:
emsapss-decode-help1 mHash EM emBits == if emBits < length (mHash) + sLen
+ 16
    then False
    else emsapss-decode-help2 mHash EM emBits

```

```

emsapss-decode-help2:
emsapss-decode-help2 mHash EM emBits == if show-rightmost-bits EM 8 ≠ BC
then False
else emsapss-decode-help3 mHash EM emBits

emsapss-decode-help3:
emsapss-decode-help3 mHash EM emBits == emsapss-decode-help4 mHash (generate-maskedDB
EM emBits (length mHash)) (generate-H EM emBits (length mHash)) emBits

emsapss-decode-help4:
emsapss-decode-help4 mHash maskedDB H emBits == if take ((roundup emBits
8)*8 - emBits) maskedDB ≠ bv-prepend ((roundup emBits 8)*8 - emBits) 0 []
then False
else emsapss-decode-help5 mHash maskedDB
(MGF H ((roundup emBits 8)*8 - (length mHash) - 8)) H emBits

emsapss-decode-help5:
emsapss-decode-help5 mHash maskedDB dbMask H emBits == emsapss-decode-help6
mHash (bxor maskedDB dbMask) H emBits

emsapss-decode-help6:
emsapss-decode-help6 mHash DB H emBits == emsapss-decode-help7 mHash
(maskedDB-zero DB emBits) H emBits

emsapss-decode-help7:
emsapss-decode-help7 mHash DB-zero H emBits == if (take ( (roundup emBits
8)*8 - (length mHash) - sLen - 16) DB-zero ≠ bv-prepend ( (roundup emBits
8)*8 - (length mHash) - sLen - 16) 0 []) ∨ (take 8 ( drop ((roundup emBits
8)*8 - (length mHash) - sLen - 16 ) DB-zero ) ≠ [Zero, Zero, Zero, Zero, Zero,
Zero, Zero, One])
then False
else emsapss-decode-help8 mHash DB-zero H

emsapss-decode-help8:
emsapss-decode-help8 mHash DB-zero H == emsapss-decode-help9 mHash (generate-salt
DB-zero) H

emsapss-decode-help9:
emsapss-decode-help9 mHash salt-new H == emsapss-decode-help10 (generate-M'
mHash salt-new) H

emsapss-decode-help10:
emsapss-decode-help10 M' H == emsapss-decode-help11 (sha1 M') H

emsapss-decode-help11:
emsapss-decode-help11 H' H == if H' ≠ H
then False
else True

```

**primrec**

$MGF2\ Z\ 0 = sha1\ (Z@(nat-to-bv-length\ 0\ 32))$

$MGF2\ Z\ (Suc\ n) = (MGF2\ Z\ n)@(sha1\ (Z@(nat-to-bv-length\ (Suc\ n)\ 32)))$

**lemma** *roundup-positiv*:  $0 < emBits \implies 0 < (roundup\ emBits\ 160)$

**by** (*simp add: roundup, safe, simp*)

**lemma** *roundup-ge-emBits*:  $0 < emBits \implies 0 < x \implies emBits \leq (roundup\ emBits\ x) * x$

**apply** (*simp add: roundup mult-commute*)

**apply** (*safe*)

**apply** (*simp*)

**apply** (*simp add: add-commute [of x x\*(emBits div x)]*)

**apply** (*insert mod-div-equality2 [of x emBits]*)

**apply** (*subgoal-tac emBits mod x < x*)

**apply** (*arith*)

**apply** (*simp only: mod-less-divisor*)

**done**

**lemma** *roundup-ge-0*:  $0 < emBits \implies 0 < x \implies 0 \leq roundup\ emBits\ x * x - emBits$

**by** (*simp add: roundup*)

**lemma** *roundup-le-7*:  $0 < emBits \implies roundup\ emBits\ 8 * 8 - emBits \leq 7$

**apply** (*simp add: roundup*)

**apply** (*insert div-mod-equality [of emBits 8 1]*)

**apply** *arith*

**done**

**lemma** *roundup-nat-ge-8-help*:

$length\ (sha1\ M) + sLen + 16 \leq emBits \implies 8 \leq roundup\ emBits\ 8 * 8 - (length\ (sha1\ M) + 8)$

**apply** (*insert roundup-ge-emBits [of emBits 8]*)

**apply** (*simp add: roundup sha1len sLen*)

**done**

**lemma** *roundup-nat-ge-8*:

$length\ (sha1\ M) + sLen + 16 \leq emBits \implies 8 \leq roundup\ emBits\ 8 * 8 - (length\ (sha1\ M) + 8)$

**apply** (*insert roundup-nat-ge-8-help [of M emBits]*)

**apply** *arith*

**done**

**lemma** *roundup-le-ub*:

$\llbracket 176 + sLen \leq emBits; emBits \leq 2^{32} * 160 \rrbracket \implies (roundup\ emBits\ 8) * 8 - 168 \leq 2^{32} * 160$

**apply** (*simp add: roundup*)

**apply** (*safe*)

**apply** (*simp*)  
**apply** (*arith*)  
**done**

**lemma** *modify-roundup-ge1*:  $\llbracket 8 \leq \text{roundup } \text{emBits } 8 * 8 - 168 \rrbracket \implies 176 \leq \text{roundup } \text{emBits } 8 * 8$   
**by** *arith*

**lemma** *modify-roundup-ge2*:  $\llbracket 176 \leq \text{roundup } \text{emBits } 8 * 8 \rrbracket \implies 21 < \text{roundup } \text{emBits } 8$   
**by** *simp*

**lemma** *roundup-help1*:  $\llbracket 0 < \text{roundup } l \ 160 \rrbracket \implies (\text{roundup } l \ 160 - 1) + 1 = (\text{roundup } l \ 160)$   
**by** *arith*

**lemma** *roundup-help1-new*:  $\llbracket 0 < l \rrbracket \implies (\text{roundup } l \ 160 - 1) + 1 = (\text{roundup } l \ 160)$   
**apply** (*drule* *roundup-positiv* [*of l*])  
**apply** *arith*  
**done**

**lemma** *roundup-help2*:  $\llbracket 176 + \text{sLen} \leq \text{emBits} \rrbracket \implies \text{roundup } \text{emBits } 8 * 8 - \text{emBits} \leq \text{roundup } \text{emBits } 8 * 8 - 160 - \text{sLen} - 16$   
**by** (*simp* *add: sLen*)

**lemma** *bv-prepend-equal*:  $\text{bv-prepend } (\text{Suc } n) \ b \ l = \text{b\#bv-prepend } n \ b \ l$   
**by** (*simp* *add: bv-prepend*)

**lemma** *length-bv-prepend*:  $\text{length } (\text{bv-prepend } n \ b \ l) = n + \text{length } l$   
**by** (*induct* *n*) (*simp-all* *add: bv-prepend*)

**lemma** *length-bv-prepend-drop*:  $a \leq \text{length } xs \longrightarrow \text{length } (\text{bv-prepend } a \ b \ (\text{drop } a \ xs)) = \text{length } xs$   
**by** (*simp* *add: length-bv-prepend*)

**lemma** *take-bv-prepend*:  $\text{take } n \ (\text{bv-prepend } n \ b \ x) = \text{bv-prepend } n \ b \ []$   
**by** (*induct* *n*) (*simp* *add: bv-prepend*)

**lemma** *take-bv-prepend2*:  $\text{take } n \ (\text{bv-prepend } n \ b \ xs @ ys @ zs) = \text{bv-prepend } n \ b \ []$   
**by** (*induct* *n*) (*simp* *add: bv-prepend*)

**lemma** *bv-prepend-append*:  $\text{bv-prepend } a \ b \ x = \text{bv-prepend } a \ b \ [] @ x$   
**by** (*induct* *a*) (*simp* *add: bv-prepend*, *simp* *add: bv-prepend-equal*)

**lemma** *bv-prepend-append2*:  
 $x < y \implies \text{bv-prepend } y \ b \ xs = (\text{bv-prepend } x \ b \ []) @ (\text{bv-prepend } (y - x) \ b \ []) @ xs$   
**by** (*simp* *add: bv-prepend replicate-add* [*symmetric*])

**lemma** *drop-bv-prepend-help2*:  $\llbracket x < y \rrbracket \implies \text{drop } x \text{ (bv-prepend } y \text{ b [])} = \text{bv-prepend (y-x) b []}$

**apply** (*insert bv-prepend-append2 [of x y b []]*)  
**by** (*simp add: length-bv-prepend*)

**lemma** *drop-bv-prepend-help3*:  $\llbracket x = y \rrbracket \implies \text{drop } x \text{ (bv-prepend } y \text{ b [])} = \text{bv-prepend (y-x) b []}$

**apply** (*insert length-bv-prepend [of y b []]*)  
**by** (*simp add: bv-prepend*)

**lemma** *drop-bv-prepend-help4*:  $\llbracket x \leq y \rrbracket \implies \text{drop } x \text{ (bv-prepend } y \text{ b [])} = \text{bv-prepend (y-x) b []}$

**apply** (*insert drop-bv-prepend-help2 [of x y b] drop-bv-prepend-help3 [of x y b]*)  
**by** (*arith*)

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

**by** (*induct x*) (*simp add: bv-prepend*)+

**lemma** *bv-prepend-drop*:  $x \leq y \implies \text{bv-prepend } x \text{ b (drop } x \text{ (bv-prepend } y \text{ b []))} = \text{bv-prepend } y \text{ b []}$

**apply** (*simp add: drop-bv-prepend-help4 [of x y b]*)  
**by** (*simp add: bv-prepend-append [of x b (bv-prepend (y - x) b [])] bv-prepend-add*)

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

**by** (*induct x*) (*simp add: bv-prepend*)+

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

**by** (*simp add: generate-DB sLen*)

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

**by** (*simp add: generate-PS length-bv-prepend*)

**lemma** *length-bvxor*:  $\text{length } a = \text{length } b \implies \text{length (bv xor } a \text{ b)} = \text{length } a$

**by** (*simp add: vxor*)

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

**by** (*induct m*) (*simp+, simp add: sha1len*)

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

**by** (*simp add: MGF1 length-MGF2 sha1len*)

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

**apply** (*simp add: MGF sha1len*)

**apply** (*insert roundup-help1-new [of l]*)

**apply** (*rule length-MGF1*)

**apply** (*simp*)  
**apply** (*insert roundup-ge-emBits [of l 160]*)  
**apply** (*arith*)  
**done**

**lemma** *solve-length-generate-DB*:

$\llbracket 0 < emBits; length (sha1 M) + sLen + 16 \leq emBits \rrbracket$   
 $\implies length (generate-DB (generate-PS emBits (length (sha1 x)))) = (roundup emBits 8) * 8 - 168$   
**apply** (*insert roundup-ge-emBits [of emBits 8]*)  
**apply** (*simp add: length-generate-DB length-generate-PS sha1len*)  
**done**

**lemma** *length-maskedDB-zero*:

$\llbracket roundup emBits 8 * 8 - emBits \leq length maskedDB \rrbracket$   
 $\implies length (maskedDB-zero maskedDB emBits) = length maskedDB$   
**by** (*simp add: maskedDB-zero length-bv-prepend*)

**lemma** *take-equal-bv-prepend*:

$\llbracket 176 + sLen \leq emBits; roundup emBits 8 * 8 - emBits \leq 7 \rrbracket$   
 $\implies take (roundup emBits 8 * 8 - length (sha1 M) - sLen - 16) (maskedDB-zero (generate-DB (generate-PS emBits 160))) emBits =$   
 $bv-prepend (roundup emBits 8 * 8 - length (sha1 M) - sLen - 16) \mathbf{0}$   
**apply** (*insert roundup-help2 [of emBits] length-generate-PS [of emBits]*)  
**apply** (*simp add: sha1len maskedDB-zero generate-DB generate-PS bv-prepend-split bv-prepend-drop*)  
**done**

**lemma** *lastbits-BC*:  $BC = show-rightmost-bits (xs @ ys @ BC) 8$

**by** (*simp add: show-rightmost-bits BC*)

**lemma** *equal-zero*:

$176 + sLen \leq emBits \implies roundup emBits 8 * 8 - emBits \leq roundup emBits 8 * 8 - (176 + sLen)$   
 $\implies 0 = roundup emBits 8 * 8 - emBits - (roundup emBits 8 * 8 - (176 + sLen))$   
**by** *arith*

**lemma** *get-salt*:  $\llbracket 176 + sLen \leq emBits; roundup emBits 8 * 8 - emBits \leq 7 \rrbracket \implies (generate-salt (maskedDB-zero (generate-DB (generate-PS emBits 160))) emBits) = salt$

**apply** (*insert roundup-help2 [of emBits] length-generate-PS [of emBits] equal-zero [of emBits]*)  
**apply** (*simp add: generate-DB generate-PS maskedDB-zero*)  
**apply** (*simp add: bv-prepend-split bv-prepend-drop generate-salt show-rightmost-bits sLen*)  
**done**

**lemma** *generate-maskedDB-elim*:  $\llbracket roundup emBits 8 * 8 - emBits \leq length x; \rrbracket$

$(\text{roundup } emBits \ 8) * 8 - (\text{length } (sha1 \ M)) - 8 = \text{length } (\text{maskedDB-zero } x \ emBits)$   
 $\implies \text{generate-maskedDB } (\text{maskedDB-zero } x \ emBits \ @ \ y \ @ \ z) \ emBits$   
 $(\text{length}(sha1 \ M)) = \text{maskedDB-zero } x \ emBits$   
**apply** (simp add: maskedDB-zero)  
**apply** (insert length-bv-prepend-drop [of (roundup emBits 8 \* 8 - emBits) x])  
**apply** (simp add: generate-maskedDB)  
**done**

**lemma** generate-H-elim:  $\llbracket \text{roundup } emBits \ 8 * 8 - emBits \leq \text{length } x; \text{length } (\text{maskedDB-zero } x \ emBits) = (\text{roundup } emBits \ 8) * 8 - 168; \text{length } y = 160 \rrbracket$   
 $\implies \text{generate-H } (\text{maskedDB-zero } x \ emBits \ @ \ y \ @ \ z) \ emBits \ 160 = y$   
**apply** (simp add: maskedDB-zero)  
**apply** (insert length-bv-prepend-drop [of roundup emBits 8 \* 8 - emBits x])  
**apply** (simp add: generate-H)  
**done**

**lemma** length-bv-prepend-drop-special:  $\llbracket \text{roundup } emBits \ 8 * 8 - emBits \leq \text{roundup } emBits \ 8 * 8 - (176 + sLen); \text{length } (\text{generate-PS } emBits \ 160) = \text{roundup } emBits \ 8 * 8 - (176 + sLen) \rrbracket \implies \text{length } (\text{bv-prepend } (\text{roundup } emBits \ 8 * 8 - emBits) \ \mathbf{0} \ (\text{drop } (\text{roundup } emBits \ 8 * 8 - emBits) \ (\text{generate-PS } emBits \ 160))) = \text{length } (\text{generate-PS } emBits \ 160)$   
**by** (simp add: length-bv-prepend-drop)

**lemma** x01-elim:  $\llbracket 176 + sLen \leq emBits; \text{roundup } emBits \ 8 * 8 - emBits \leq 7 \rrbracket \implies \text{take } 8 \ (\text{drop } (\text{roundup } emBits \ 8 * 8 - (\text{length } (sha1 \ M) + sLen + 16)) (\text{maskedDB-zero } (\text{generate-DB } (\text{generate-PS } emBits \ 160)) \ emBits)) = [\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{1}]$   
**apply** (insert roundup-help2 [of emBits] length-generate-PS [of emBits] equal-zero [of emBits])  
**apply** (simp add: sha1len maskedDB-zero generate-DB generate-PS bv-prepend-split bv-prepend-drop)  
**done**

**lemma** drop-bv-mapzip:  
**assumes**  $n \leq \text{length } x \ \text{length } x = \text{length } y$   
**shows**  $\text{drop } n \ (\text{bv-mapzip } f \ x \ y) = \text{bv-mapzip } f \ (\text{drop } n \ x) \ (\text{drop } n \ y)$   
**proof** -  
**have**  $\bigwedge x \ y. \ n \leq \text{length } x \implies \text{length } x = \text{length } y \implies \text{drop } n \ (\text{bv-mapzip } f \ x \ y) = \text{bv-mapzip } f \ (\text{drop } n \ x) \ (\text{drop } n \ y)$   
**apply** (induct n)  
**apply** simp  
**apply** (case-tac x, case-tac[!] y, auto)  
**done**  
**with** assms **show** ?thesis **by** simp  
**qed**

**lemma** [simp]:  
**assumes**  $\text{length } a = \text{length } b$   
**shows**  $\text{bxor } (\text{bxor } a \ b) \ b = a$

```

proof –
  have  $\wedge b. \text{length } a = \text{length } b \implies \text{bxor } (\text{bxor } a \ b) \ b = a$ 
    apply (induct a)
    apply (auto simp add: bxor)
    apply (case-tac b)
    apply (simp)+
    apply (case-tac a1)
    apply (case-tac a)
    apply (safe)
    apply (simp)+
    apply (case-tac a)
    apply (simp)+
    done
  with assms show ?thesis by simp
qed

lemma bxorxor-elim-help:
  assumes  $x \leq \text{length } a$  and  $\text{length } a = \text{length } b$ 
  shows  $\text{bv-prepend } x \ \mathbf{0} \ (\text{drop } x \ (\text{bxor } (\text{bv-prepend } x \ \mathbf{0} \ (\text{drop } x \ (\text{bxor } a \ b)))) \ b))$ 
  =
     $\text{bv-prepend } x \ \mathbf{0} \ (\text{drop } x \ a)$ 
proof –
  have  $\text{drop } x \ (\text{bxor } (\text{bv-prepend } x \ \mathbf{0} \ (\text{drop } x \ (\text{bxor } a \ b))) \ b) = \text{drop } x \ a$ 
    apply (unfold bxor bv-prepend)
    apply (cut-tac assms)
    apply (insert length-replicate [of x 0])
    apply (insert length-drop [of x a])
    apply (insert length-drop [of x b])
    apply (insert length-bxor [of drop x a drop x b])
    apply (subgoal-tac length (replicate x 0 @ drop x (bv-mapzip op ⊕b a b)) =
length b)
    apply (subgoal-tac b = (take x b)@(drop x b))
    apply (insert drop-bv-mapzip [of x (replicate x 0 @ drop x (bv-mapzip op ⊕b a
b)) b op ⊕b])
    apply (simp)
    apply (insert drop-bv-mapzip [of x a b op ⊕b])
    apply (simp)
    apply (fold bxor)
    apply (simp-all)
    done
  with assms show ?thesis by simp
qed

lemma bxorxor-elim:  $\llbracket \text{roundup } \text{emBits } 8 * 8 - \text{emBits} \leq \text{length } a; \text{length } a = \text{length } b \rrbracket \implies (\text{maskedDB-zero } (\text{bxor } (\text{maskedDB-zero } (\text{bxor } a \ b) \ \text{emBits}) \ b) \ \text{emBits}) = \text{bv-prepend } (\text{roundup } \text{emBits } 8 * 8 - \text{emBits}) \ \mathbf{0} \ (\text{drop } (\text{roundup } \text{emBits } 8 * 8 - \text{emBits}) \ a)$ 
  by (simp add: maskedDB-zero bxorxor-elim-help)

```

```

lemma verify:  $\llbracket (\text{emsapss-encode } M \text{ emBits}) \neq []; EM=(\text{emsapss-encode } M \text{ em-}
\text{Bits}) \rrbracket \implies \text{emsapss-decode } M \text{ EM emBits} = \text{True}$ 
  apply (simp add: emsapss-decode emsapss-encode)
  apply (safe, simp+)
  apply (simp add: emsapss-decode-help1 emsapss-encode-help1)
  apply (safe, simp+)
  apply (simp add: emsapss-decode-help2 emsapss-encode-help2)
  apply (safe)
  apply (simp add: emsapss-encode-help3 emsapss-encode-help4 emsapss-encode-help5
emsapss-encode-help6)
  apply (safe)
  apply (simp add: emsapss-encode-help7 emsapss-encode-help8 lastbits-BC [symmetric])+
  apply (simp add: emsapss-decode-help3 emsapss-encode-help3 emsapss-decode-help4
emsapss-encode-help4)
  apply (safe)
  apply (insert roundup-le-7 [of emBits] roundup-ge-0 [of emBits 8] roundup-nat-ge-8
[of M emBits])
  apply (simp add: generate-maskedDB emsapss-encode-help5 emsapss-encode-help6)
  apply (safe)
  apply (simp)
  apply (simp add: emsapss-encode-help7)
  apply (simp only: emsapss-encode-help8)
  apply (simp only: maskedDB-zero)
  apply (simp only: take-bv-prepend2 min-max.inf-absorb1)
  apply (simp)
  apply (simp add: emsapss-encode-help5 emsapss-encode-help6)
  apply (safe)
  apply (simp)+
  apply (insert solve-length-generate-DB [of emBits M generate-M' (sha1 M) salt]
roundup-le-ub [of emBits])
  apply (insert length-MGF [of (roundup emBits 8) * 8 - 168 (sha1 (generate-M'
(sha1 M) salt))])
  apply (insert modify-roundup-ge1 [of emBits] modify-roundup-ge2 [of emBits])
  apply (simp add: sha1len emsapss-encode-help7 emsapss-encode-help8)
  apply (insert length-bv xor [of (generate-DB (generate-PS emBits 160)) (MGF
(sha1 (generate-M' (sha1 M) salt)) ((roundup emBits 8) * 8 - 168))])
  apply (insert generate-maskedDB-elim [of emBits (bv xor (generate-DB (generate-PS
emBits 160))(MGF (sha1 (generate-M' (sha1 M) salt)) ((roundup emBits 8) * 8
- 168))) M sha1 (generate-M' (sha1 M) salt) BC])
  apply (insert length-maskedDB-zero [of emBits (bv xor (generate-DB (generate-PS
emBits 160))(MGF (sha1 (generate-M' (sha1 M) salt)) ((roundup emBits 8) * 8
- 168))])
  apply (insert generate-H-elim [of emBits (bv xor (generate-DB (generate-PS em-
Bits 160))(MGF (sha1 (generate-M' (sha1 M) salt)) (roundup emBits 8 * 8 -
168))) sha1 (generate-M' (sha1 M) salt) BC])
  apply (simp add: sha1len emsapss-decode-help5)
  apply (simp only: emsapss-decode-help6 emsapss-decode-help7)
  apply (insert bv xor xor-elim [of emBits (generate-DB (generate-PS emBits 160))
(MGF (sha1 (generate-M' (sha1 M) salt)) ((roundup emBits 8) * 8 - 168))])

```

```

apply (fold maskedDB-zero)
apply (insert take-equal-bv-prepend [of emBits M] x01-elim [of emBits M] get-salt
[of emBits])
apply (simp add: emsapss-decode-help8 emsapss-decode-help9 emsapss-decode-help10
emsapss-decode-help11)
done

end

```

### 13 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

```

consts rsapss-sign:: bv  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bv
         rsapss-sign-help1:: nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bv
         rsapss-verify:: bv  $\Rightarrow$  bv  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bool

```

```

defs

```

```

rsapss-sign:
rsapss-sign m e n == if (emsapss-encode m (length (nat-to-bv n) - 1)) = []
                        then []
                        else (rsapss-sign-help1 (bv-to-nat (emsapss-encode m (length
(nat-to-bv n) - 1))) e n)

```

```

rsapss-sign-help1:
rsapss-sign-help1 em-nat e n == nat-to-bv-length (rsa-crypt em-nat e n) (length
(nat-to-bv n))

```

```

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

```

```

lemma length-emsapss-encode:

```

```

emsapss-encode m x  $\neq$  []  $\implies$  length (emsapss-encode m x) = roundup x 8 * 8

```

```

apply (atomize (full))
apply (simp add: emsapss-encode)
apply (simp add: emsapss-encode-help1)
apply (simp add: emsapss-encode-help2)
apply (simp add: emsapss-encode-help3)
apply (simp add: emsapss-encode-help4)

```

```

apply (simp add: emsapss-encode-help5)
apply (simp add: emsapss-encode-help6)
apply (simp add: emsapss-encode-help7)
apply (simp add: emsapss-encode-help8)
apply (simp add: maskedDB-zero)
apply (simp add: length-generate-DB)
apply (simp add: sha1len)
apply (simp add: bvxor)
apply (simp add: length-generate-PS)
apply (simp add: length-bv-prepend)
apply (simp add: MGF)
apply (simp add: MGF1)
apply (simp add: length-MGF2)
apply (simp add: sha1len)
apply (simp add: length-generate-DB)
apply (simp add: length-generate-PS)
apply (simp add: BC)
apply (insert roundup-ge-emBits [of x 8])
apply safe
apply (simp add: min-max.sup-absorb1)
done

```

**lemma** *bv-to-nat-emsapss-encode-le*:  $\text{emsapss-encode } m \ x \neq [] \implies \text{bv-to-nat } (\text{emsapss-encode } m \ x) < 2^{\text{roundup } x \ 8 \ * \ 8}$

```

apply (insert length-emsapss-encode [of m x])
apply (insert bv-to-nat-upper-range [of emsapss-encode m x])
by (simp)

```

**lemma** *length-helper1*: **shows** *length*

```

(bvxor
 (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
 (bxor
 (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** –  
**have**  $a$ :  $\text{length } BC = 8$  **by** (*simp add: BC*)  
**have**  $b$ :  $\text{length } (\text{sha1 } (\text{generate-}M' (\text{sha1 } m) \text{ salt})) = 160$  **by** (*simp add: sha1len*)  
**have**  $c$ :  $\bigwedge a b c. \text{length } (a@b@c) = \text{length } a + \text{length } b + \text{length } c$  **by** *simp*  
**from**  $a$  **and**  $b$  **show** *?thesis* **using**  $c$  **by** *simp*  
**qed**

**lemma** *MGFLen-helper*:  $MGF\ z\ l \sim = [] \implies l \leq 2^{32} * (\text{length } (\text{sha1 } z))$

**proof** (*cases*  $2^{32} * \text{length } (\text{sha1 } z) < l$ )  
**assume**  $x$ :  $MGF\ z\ l \sim = []$   
**assume**  $a$ :  $2^{32} * \text{length } (\text{sha1 } z) < l$   
**then have**  $MGF\ z\ l = []$   
**proof** (*cases*  $l=0$ )  
**assume**  $l=0$   
**then show**  $MGF\ z\ l = []$  **by** (*simp add: MGF*)  
**next**  
**assume**  $l \neq 0$   
**then have**  $(l = 0 \vee 2^{32} * \text{length } (\text{sha1 } z) < l) = \text{True}$  **using**  $a$  **by** *fast*  
**then show**  $MGF\ z\ l = []$  **apply** (*simp only: MGF*) **by** *simp*  
**qed**  
**then show** *?thesis* **using**  $x$  **by** *simp*  
**next**  
**assume**  $\neg 2^{32} * \text{length } (\text{sha1 } z) < l$   
**then show** *?thesis* **by** *simp*  
**qed**

**lemma** *length-helper2*: **assumes**  $p$ : *prime*  $p$  **and**  $q$ : *prime*  $q$

**and**  $mgf$ :  $(MGF (\text{sha1 } (\text{generate-}M' (\text{sha1 } m) \text{ salt})))$

(*length*  
*generate-DB*  
*generate-PS* (*length* (*nat-to-bv* ( $p * q$ )) – *Suc* 0)  
(*length* (*sha1* (*generate-}M' (\text{sha1 } m) \text{ salt})))))) \sim = []  
**and**  $len$ :  $\text{length } (\text{sha1 } M) + sLen + 16 \leq (\text{length } (\text{nat-to-bv } (p * q))) - \text{Suc } 0$   
**shows** *length*  
(  
(*bxor*  
*generate-DB*  
*generate-PS* (*length* (*nat-to-bv* ( $p * q$ )) – *Suc* 0)  
(*length* (*sha1* (*generate-}M' (\text{sha1 } m) \text{ salt}))))))  
(*MGF* (*sha1* (*generate-}M' (\text{sha1 } m) \text{ salt})))  
(*length*  
*generate-DB*  
*generate-PS* (*length* (*nat-to-bv* ( $p * q$ )) – *Suc* 0)  
(*length* (*sha1* (*generate-}M' (\text{sha1 } m) \text{ salt}))))))  
) = (*roundup* (*length* (*nat-to-bv* ( $p * q$ )) – *Suc* 0) 8) \* 8 – 168****

**proof** –  
**have**  $a$ :  $\text{length } (MGF (\text{sha1 } (\text{generate-}M' (\text{sha1 } m) \text{ salt})))$   
(*length*  
*generate-DB*

```

(generate-PS (length (nat-to-bv (p * q)) - Suc 0)
(length (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))))))
proof -
  have 0 < (length
(generate-DB
(generate-PS (length (nat-to-bv (p * q)) - Suc 0)
(length (sha1 (generate-M' (sha1 m) salt)))))) by (simp add: generate-DB)
  moreover have (length
(generate-DB
(generate-PS (length (nat-to-bv (p * q)) - Suc 0)
(length (sha1 (generate-M' (sha1 m) salt)))))) ≤ 232 * length (sha1 (sha1
(generate-M' (sha1 m) salt))) using mgf and MGFLen-helper by simp
  ultimately show ?thesis using length-MGF by simp
qed
have b: 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)
proof -
  have 0 ≤ (length (nat-to-bv (p * q))) - Suc 0
proof -
  from p have p2: 1 < p by (simp add: prime-def)
  moreover from q have 1 < q by (simp add: prime-def)
  ultimately have p < p*q by simp
  then have 1 < p*q using p2 by arith
  then show ?thesis using len-nat-to-bv-pos by simp
qed
then show ?thesis using solve-length-generate-DB using len by simp
qed
have c: length (bvxor
(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 using a and b and
length-bvxor by simp
then show ?thesis by simp
qed

lemma emBits-roundup-cancel: emBits mod 8 ~ = 0 ⇒ (roundup emBits 8)*8
- emBits = 8-(emBits mod 8)
apply (auto simp add: roundup)

```

by (arith)

**lemma** *emBits-roundup-cancel2*:  $emBits \bmod 8 \sim=0 \implies (roundup\ emBits\ 8) * 8 - (8 - (emBits \bmod 8)) = emBits$   
 by (auto simp add: roundup)

**lemma** *length-bound*:  $\llbracket emBits \bmod 8 \sim=0; 8 \leq (length\ maskedDB) \rrbracket \implies length\ (remzero\ ((maskedDB-zero\ maskedDB\ emBits)@a@b)) \leq length\ (maskedDB@a@b) - (8 - (emBits \bmod 8))$

**proof** -

assume *a*:  $emBits \bmod 8 \sim=0$

assume *len*:  $8 \leq (length\ maskedDB)$

have *b*:  $\bigwedge a. length\ (remzero\ a) \leq length\ a$

**proof** -

fix *a*

show  $length\ (remzero\ a) \leq length\ a$

**proof** (induct *a*)

show  $(length\ (remzero\ [])) \leq length\ []$  by (simp)

next

case (Cons *hd tl*)

show  $(length\ (remzero\ (hd\#\#tl))) \leq length\ (hd\#\#tl)$

**proof** (cases *hd*)

assume *hd*=0

then have  $remzero\ (hd\#\#tl) = remzero\ tl$  by simp

then show ?thesis using Cons by simp

next

assume *hd*=1

then have  $remzero\ (hd\#\#tl) = hd\#\#tl$  by simp

then show ?thesis by simp

qed

qed

from *len* show  $length\ (remzero\ (maskedDB-zero\ maskedDB\ emBits\ @\ a\ @\ b)) \leq length\ (maskedDB\ @\ a\ @\ b) - (8 - emBits \bmod 8)$

**proof** -

have  $remzero(bv-prepend\ ((roundup\ emBits\ 8) * 8 - emBits))$

$0\ (drop\ ((roundup\ emBits\ 8)*8 - emBits)\ maskedDB)@a@b = remzero\ ((drop\ ((roundup\ emBits\ 8)*8 - emBits)\ maskedDB)@a@b)$  using *remzero-replicate* by (simp add: *bv-prepend*)

moreover from *emBits-roundup-cancel* have  $roundup\ emBits\ 8 * 8 - emBits = 8 - emBits \bmod 8$  using *a* by simp

moreover have  $length\ ((drop\ (8 - emBits \bmod 8)\ maskedDB)@a@b) = length\ (maskedDB@a@b) - (8 - emBits \bmod 8)$

**proof** -

show ?thesis using *length-drop*[of  $(8 - emBits \bmod 8)\ maskedDB$ ]

**proof** (simp)

have  $0 \leq emBits \bmod 8$  by simp

then have  $8 - (emBits \bmod 8) \leq 8$  by simp

then show  $length\ maskedDB + emBits \bmod 8 - 8 + (length\ a + length$

$b) =$   
 $\text{length maskedDB} + (\text{length } a + \text{length } b) + \text{emBits mod } 8 - 8$  **using**  $\text{len}$   
**by** *arith*  
**qed**  
**qed**  
**ultimately show** *?thesis* **using**  $b[\text{of } (\text{drop } ((\text{roundup } \text{emBits } 8) * 8 - \text{emBits})$   
 $\text{maskedDB}) @ a @ b]$  **by** (*simp add: maskedDB-zero*)  
**qed**  
**qed**

**lemma** *length-bound2*:  $8 \leq \text{length } ( \text{bvxor}$   
 $(\text{generate-DB}$   
 $(\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0)$   
 $(\text{length } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}))))))$   
 $(\text{MGF } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt})))$   
 $(\text{length}$   
 $(\text{generate-DB}$   
 $(\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0)$   
 $(\text{length } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}))))))$

**proof** –  
**have**  $8 \leq \text{length } (\text{generate-DB}$   
 $(\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0)$   
 $(\text{length } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}))))))$  **by** (*simp add: generate-DB*)  
**then show** *?thesis* **using** *length-bvxor-bound* **by** *simp*  
**qed**

**lemma** *length-helper*: **assumes**  $p$ : prime  $p$  **and**  $q$ : prime  $q$  **and**  $x$ :  $(\text{length } (\text{nat-to-bv}$   
 $(p * q)) - \text{Suc } 0) \text{ mod } 8 \sim = 0$  **and** *mgf*:  $(\text{MGF } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m)$   
 $\text{salt})))$   
 $(\text{length}$   
 $(\text{generate-DB}$   
 $(\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0)$   
 $(\text{length } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt})))))) \sim = []$   
**and**  $\text{len: length } (\text{sha1 } M) + \text{sLen} + 16 \leq (\text{length } (\text{nat-to-bv } (p * q))) - \text{Suc } 0$   
**shows** *length*  
 $(\text{remzero}$   
 $(\text{maskedDB-zero}$   
 $(\text{bvxor}$   
 $(\text{generate-DB}$   
 $(\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0)$   
 $(\text{length } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}))))))$   
 $(\text{MGF } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt})))$   
 $(\text{length}$   
 $(\text{generate-DB}$   
 $(\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0)$   
 $(\text{length } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}))))))$   
 $(\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0) @$   
 $\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}) @ \text{BC}))$   
 $< \text{length } (\text{nat-to-bv } (p * q))$

**proof** –  
**from** *mgf* **have** *round*:  $168 \leq \text{roundup} (\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0) 8 * 8$   
**proof** (*simp only: sha1len sLen*)  
**from** *len* **have**  $160 + \text{sLen} + 16 \leq \text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0$  **by** (*simp add: sha1len*)  
**then** **have** *len1*:  $176 \leq \text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0$  **by** *simp*  
**have**  $\text{length} (\text{nat-to-bv} (p*q)) - \text{Suc } 0 \leq (\text{roundup} (\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0) 8) * 8$   
**unfolding** *roundup*  
**proof** (*cases* ( $\text{length} (\text{nat-to-bv} (p*q)) - \text{Suc } 0 \bmod 8 = 0$ )  
**assume** *len2*: ( $\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0 \bmod 8 = 0$ )  
**then** **have** (*if* ( $\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0 \bmod 8 = 0$ ) *then* ( $\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0 \text{ div } 8$ ) *else* ( $\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0 \text{ div } 8 + 1$ )  $* 8 = (\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0) \text{ div } 8 * 8$ ) **by** *simp*  
**moreover** **have** ( $\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0 \text{ div } 8 * 8 = (\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0)$ ) **using** *len2* **by** (*auto simp add: div-mod-equality[of length (nat-to-bv (p \* q)) - Suc 0 8 0]*)  
**ultimately show**  $\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0 \leq (\text{if} (\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0 \bmod 8 = 0 \text{ then } (\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0) \text{ div } 8 \text{ else } (\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0) \text{ div } 8 + 1) * 8$   
**by** *simp*  
**next**  
**assume** *len2*: ( $\text{length} (\text{nat-to-bv} (p*q)) - \text{Suc } 0 \bmod 8 \sim = 0$ )  
**then** **have** (*if* ( $\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0 \bmod 8 = 0$ ) *then* ( $\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0 \text{ div } 8$ ) *else* ( $\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0 \text{ div } 8 + 1$ )  $* 8 = ((\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0) \text{ div } 8 + 1) * 8$ ) **by** *simp*  
**moreover** **have**  $\text{length} (\text{nat-to-bv} (p*q)) - \text{Suc } 0 \leq ((\text{length} (\text{nat-to-bv} (p*q)) - \text{Suc } 0) \text{ div } 8 + 1) * 8$  **by** *auto*  
**ultimately show**  $\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0 \leq (\text{if} (\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0 \bmod 8 = 0 \text{ then } (\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0) \text{ div } 8 \text{ else } (\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0) \text{ div } 8 + 1) * 8$   
**by** *simp*  
**qed**  
**then show**  $168 \leq \text{roundup} (\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0) 8 * 8$  **using** *len1* **by** *simp*  
**qed**  
**from** *x* **have** *a*: *length*  
(*remzero*)  
(*maskedDB-zero*)  
(*bvxor*)  
(*generate-DB*)  
(*generate-PS* ( $\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0$ )  
( $\text{length} (\text{sha1} (\text{generate-M}' (\text{sha1 } m) \text{ salt}))))$ )  
( $\text{MGF} (\text{sha1} (\text{generate-M}' (\text{sha1 } m) \text{ salt}))$ )  
( $\text{length}$ )  
(*generate-DB*)  
(*generate-PS* ( $\text{length} (\text{nat-to-bv} (p * q)) - \text{Suc } 0$ )  
( $\text{length} (\text{sha1} (\text{generate-M}' (\text{sha1 } m) \text{ salt}))))$ ))))))

$(\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0) @$   
 $\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}) @ BC) <= \text{length } ((\text{bxor}$   
 $(\text{generate-DB}$   
 $(\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0)$   
 $(\text{length } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt})))))) @$   
 $(\text{MGF } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt})))$   
 $(\text{length}$   
 $(\text{generate-DB}$   
 $(\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0)$   
 $(\text{length } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt})))))) @$   
 $\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}) @ BC) - (8 - (\text{length } (\text{nat-to-bv } (p * q)) -$   
 $\text{Suc } 0) \bmod 8) \text{ using length-bound and length-bound2 by simp}$   
**have**  $b$ :  $\text{length } (\text{bxor } (\text{generate-DB } (\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q)) -$   
 $\text{Suc } 0) (\text{length } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}))))))$   
 $(\text{MGF } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}))) (\text{length } (\text{generate-DB}$   
 $(\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0) (\text{length } (\text{sha1 } (\text{generate-M}'$   
 $(\text{sha1 } m) \text{ salt})))))) @$   
 $\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}) @ BC) = \text{length } (\text{bxor } (\text{generate-DB}$   
 $(\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0) (\text{length } (\text{sha1 } (\text{generate-M}'$   
 $(\text{sha1 } m) \text{ salt}))))))$   
 $(\text{MGF } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}))) (\text{length } (\text{generate-DB}$   
 $(\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0) (\text{length } (\text{sha1 } (\text{generate-M}'$   
 $(\text{sha1 } m) \text{ salt})))))) + 168 \text{ using length-helper1 by simp}$   
**have**  $c$ :  $\text{length } (\text{bxor } (\text{generate-DB } (\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q)) -$   
 $\text{Suc } 0) (\text{length } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}))))))$   
 $(\text{MGF } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}))) (\text{length } (\text{generate-DB } (\text{generate-PS}$   
 $(\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0) (\text{length } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}))))))$   
 $=$   
 $(\text{roundup } (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0) 8) * 8 - 168 \text{ using } p$   
**and**  $q$  **and**  $\text{length-helper2}$  **and**  $\text{mgf}$  **and**  $\text{len}$  **by simp**  
**from**  $a$  **and**  $b$  **and**  $c$  **have**  $\text{length } (\text{remzero } (\text{maskedDB-zero}$   
 $(\text{bxor } (\text{generate-DB } (\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q)) -$   
 $\text{Suc } 0) (\text{length } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}))))))$   
 $(\text{MGF } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt})))$   
 $(\text{length } (\text{generate-DB } (\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q))$   
 $- \text{Suc } 0) (\text{length } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt})))))) @$   
 $\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}) @ BC) <= \text{roundup } (\text{length}$   
 $(\text{nat-to-bv } (p * q)) - \text{Suc } 0) 8 * 8 - 168 + 168 - (8 - (\text{length } (\text{nat-to-bv } (p *$   
 $q)) - \text{Suc } 0) \bmod 8) \text{ by simp}$   
**then have**  $\text{length } (\text{remzero } (\text{maskedDB-zero}$   
 $(\text{bxor } (\text{generate-DB } (\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q)) -$   
 $\text{Suc } 0) (\text{length } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}))))))$   
 $(\text{MGF } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt})))$   
 $(\text{length } (\text{generate-DB } (\text{generate-PS } (\text{length } (\text{nat-to-bv } (p * q))$   
 $- \text{Suc } 0) (\text{length } (\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt})))))) @$   
 $\text{sha1 } (\text{generate-M}' (\text{sha1 } m) \text{ salt}) @ BC) <= \text{roundup } (\text{length}$   
 $(\text{nat-to-bv } (p * q)) - \text{Suc } 0) 8 * 8 - (8 - (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0)$

$\text{mod } 8$ ) **using** *round* **by** *simp*  
**moreover have**  $\text{roundup } (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0) \ 8 * 8 - (8 - (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0) \text{ mod } 8) = (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0)$   
**using**  $x$  **and** *emBits-roundup-cancel2* **by** *simp*  
**moreover have**  $0 < \text{length } (\text{nat-to-bv } (p * q))$   
**proof** –  
**from**  $p$  **have**  $s: 1 < p$  **by** (*simp add: prime-def*)  
**moreover from**  $q$  **have**  $1 < q$  **by** (*simp add: prime-def*)  
**ultimately have**  $p < p * q$  **by** *simp*  
**then have**  $1 < p * q$  **using**  $s$  **by** *arith*  
**then show** *?thesis* **using** *len-nat-to-bv-pos* **by** *simp*  
**qed**  
**ultimately show** *?thesis* **by** *arith*  
**qed**

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

**proof** –  
**assume**  $a: \text{emsapss-encode } m \ (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0) \neq \llbracket$   
**assume**  $p: \text{prime } p$  **and**  $q: \text{prime } q$   
**assume**  $x: (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0) \text{ mod } 8 \sim = 0$   
**have**  $b: \text{emsapss-encode } m \ (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0) = \text{emsapss-encode-help1 } (\text{sha1 } m)$   
 $(\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0)$   
**proof** (*simp only: emsapss-encode*)  
**from**  $a$  **show** (*if*  $((2^{64} \leq \text{length } m) \vee (2^{32} * 160 < (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0)))$   
*then*  $\llbracket$   
*else*  $(\text{emsapss-encode-help1 } (\text{sha1 } m) \ (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0)))$   
 $= (\text{emsapss-encode-help1 } (\text{sha1 } m) \ (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0))$  **by** (*auto simp add: emsapss-encode*)  
**qed**  
**have**  $c: \text{length } (\text{remzero } (\text{emsapss-encode-help1 } (\text{sha1 } m) \ (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0))) < \text{length } (\text{nat-to-bv } (p * q))$   
**proof** (*simp only: emsapss-encode-help1*)  
**from**  $a$  **and**  $b$  **have**  $d: (\text{if } ((\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0) < (\text{length } (\text{sha1 } m) + sLen + 16)))$   
*then*  $\llbracket$   
*else*  $(\text{emsapss-encode-help2 } (\text{generate-M}' (\text{sha1 } m) \ \text{salt})$   
 $(\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0))) = (\text{emsapss-encode-help2 } ((\text{generate-M}'$   
 $(\text{sha1 } m)) \ \text{salt}) \ (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0))$  **by** (*auto simp add: emsapss-encode emsapss-encode-help1*)  
**from**  $d$  **have**  $\text{len}: \text{length } (\text{sha1 } m) + sLen + 16 \leq (\text{length } (\text{nat-to-bv } (p * q))) - \text{Suc } 0$   
**proof** (*cases*  $\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0 < \text{length } (\text{sha1 } m) + sLen + 16$ )  
**assume**  $\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0 < \text{length } (\text{sha1 } m) + sLen + 16$

```

    then have len1: (if length (nat-to-bv (p * q)) - Suc 0 < length (sha1 m)
+ sLen + 16 then []
    else emsapss-encode-help2 (generate-M' (sha1 m) salt) (length (nat-to-bv (p
* q)) - Suc 0)) = [] by simp
    assume len2: (if length (nat-to-bv (p * q)) - Suc 0 < length (sha1 m) +
sLen + 16 then []
    else emsapss-encode-help2 (generate-M' (sha1 m) salt) (length (nat-to-bv (p
* q)) - Suc 0)) =
    emsapss-encode-help2 (generate-M' (sha1 m) salt) (length (nat-to-bv (p * q))
- Suc 0)
    from len1 and len2 and a and b show length (sha1 m) + sLen +
16 ≤ length (nat-to-bv (p * q)) - Suc 0 by (auto simp add: emsapss-encode
emsapss-encode-help1)
    next
    assume ¬ length (nat-to-bv (p * q)) - Suc 0 < length (sha1 m) + sLen +
16
    then show length (sha1 m) + sLen + 16 ≤ length (nat-to-bv (p * q)) -
Suc 0 by simp
    qed
    have e: length (remzero (emsapss-encode-help2 (generate-M' (sha1 m) salt)
(length (nat-to-bv (p * q)) - Suc 0))) < length (nat-to-bv (p * q))
    proof (simp only: emsapss-encode-help2)
    show length
    (remzero
    (emsapss-encode-help3 (sha1 (generate-M' (sha1 m) salt))
    (length (nat-to-bv (p * q)) - Suc 0)))
    < length (nat-to-bv (p * q))
    proof (simp add: emsapss-encode-help3 emsapss-encode-help4 emsapss-encode-help5)
    show length
    (remzero
    (emsapss-encode-help6
    (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))
    (length (nat-to-bv (p * q)) - Suc 0)))
    < length (nat-to-bv (p * q))
    proof (simp only: emsapss-encode-help6)
    from a and b and d have mgf: 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)))))) ~ =

```

```

    [] by (auto simp add: emsapss-encode emsapss-encode-help1 emsapss-encode-help2
emsapss-encode-help3 emsapss-encode-help4 emsapss-encode-help5 emsapss-encode-help6)
    from a and b and d have f: (if 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)))))) =
    []
    then []
    else (emsapss-encode-help7
    (bvxor
    (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))
    (length (nat-to-bv (p * q)) - Suc 0))) = (emsapss-encode-help7
    (bvxor
    (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))
    (length (nat-to-bv (p * q)) - Suc 0))) by (auto simp add: emsapss-encode
emsapss-encode-help1 emsapss-encode-help2 emsapss-encode-help3 emsapss-encode-help4
emsapss-encode-help5 emsapss-encode-help6)
    have length (remzero (emsapss-encode-help7
    (bvxor
    (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))
    (length (nat-to-bv (p * q)) - Suc
0))) < length (nat-to-bv (p * q))
    proof (simp add: emsapss-encode-help7 emsapss-encode-help8)
    from p and q and x show length

```

```

      (remzero
      (maskedDB-zero
      (bxor
      (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)))))))))
      (length (nat-to-bv (p * q)) - Suc 0) @
      sha1 (generate-M' (sha1 m) salt) @ BC))
      < length (nat-to-bv (p * q)) using length-helper and len and mgf by
simp
qed
then show length
      (remzero
      (if 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)))))) =
      []
      then []
      else emsapss-encode-help7
      (bxor
      (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))
      (length (nat-to-bv (p * q)) - Suc 0)))
      < length (nat-to-bv (p * q)) using f by simp
qed
qed
qed
from d and e show length
      (remzero
      (if length (nat-to-bv (p * q)) - Suc 0 < length (sha1 m) + sLen + 16
      then []
      else emsapss-encode-help2 (generate-M' (sha1 m) salt)
      (length (nat-to-bv (p * q)) - Suc 0)))
      < length (nat-to-bv (p * q)) by simp
qed

```

from  $b$  and  $c$  show *?thesis* by *simp*  
qed

**lemma** *bv-to-nat-emsapss-smaller-pq*: **assumes**  $a$ : prime  $p$  and  $b$ : prime  $q$  and  $p \neq q$ :  $p \sim q$  and  $c$ : *emsapss-encode*  $m$  ( $\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0$ )  $\neq 0$   
**shows** *bv-to-nat* (*emsapss-encode*  $m$  ( $\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0$ ))  $< p * q$   
**proof** –

from  $a$  and  $b$  and  $c$  show *?thesis*

**proof** (*cases*  $8 \text{ dvd } ((\text{length}(\text{nat-to-bv}(p * q))) - \text{Suc } 0)$ )

assume  $d$ :  $8 \text{ dvd } ((\text{length}(\text{nat-to-bv}(p * q))) - \text{Suc } 0)$

then have  $2^{\wedge}(\text{roundup}(\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0) * 8) < p * q$

**proof** –

from  $d$  have  $e$ :  $\text{roundup}(\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0) * 8 = \text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0$  using *rnddvd* by *simp*

have  $p * q = \text{bv-to-nat}(\text{nat-to-bv}(p * q))$  by *simp*

then have  $2^{\wedge}(\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0) < p * q$

**proof** –

have  $0 < p * q$

**proof** –

have  $0 < p$  using  $a$  by (*simp add: prime-def*)

moreover have  $0 < q$  using  $b$  by (*simp add: prime-def*)

ultimately show *?thesis* by *simp*

qed

moreover have  $2^{\wedge}(\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0) \sim p * q$

**proof** (*cases*  $2^{\wedge}(\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0) = p * q$ )

assume  $2^{\wedge}(\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0) = p * q$

then have  $p = q$  using  $a$  and  $b$  and *prime-equal* by *simp*

then show *?thesis* using  $p \neq q$  by *simp*

next

assume  $2^{\wedge}(\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0) \sim p * q$

then show *?thesis* by *simp*

qed

ultimately show *?thesis* using *len-lower-bound*[of  $p * q$ ] by (*simp*)

qed

then show *?thesis* using  $e$  by *simp*

qed

moreover from  $c$  have *bv-to-nat* (*emsapss-encode*  $m$  ( $\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0$ ))  $< 2^{\wedge}(\text{roundup}(\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0) * 8)$

using *bv-to-nat-emsapss-encode-le* [of  $m$  ( $\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0$ )] by *auto*

ultimately show *?thesis* by *simp*

next

assume  $y$ :  $\sim(8 \text{ dvd } (\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0))$

then show *?thesis*

**proof** –

from  $y$  have  $x$ :  $\sim((\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0) \bmod 8 = 0)$  by (*simp add: dvd-eq-mod-eq-0*)

from *remzeroeq* have  $d$ : *bv-to-nat* (*emsapss-encode*  $m$  ( $\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0$ )) = *bv-to-nat* (*remzero* (*emsapss-encode*  $m$  ( $\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0$ )))

$q)) - \text{Suc } 0)))$  **by** *simp*  
**from** *a* **and** *b* **and** *c* **and** *x* **and** *length-emsapss-smaller-pq*[*of p q m*] **have**  
 $\text{bv-to-nat } (\text{remzero } (\text{emsapss-encode } m \ (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0)))$   
 $< \text{bv-to-nat } (\text{nat-to-bv } (p * q))$  **using** *length-lower*[*of remzero (emsapss-encode m*  
 $(\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0)) \text{ nat-to-bv } (p * q)$ ] **and** *prime-hd-non-zero*[*of*  
 $p\ q$ ] **by** (*auto*)  
**then show**  $\text{bv-to-nat } (\text{emsapss-encode } m \ (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc}$   
 $0)) < p * q$  **using** *d* **and** *bv-nat-bv* **by** *simp*  
**qed**  
**qed**  
**qed**

**lemma** *rsa-pss-verify*:  $\llbracket \text{prime } p; \text{prime } q; p \neq q; n = p * q; e * d \bmod ((\text{pred } p) * (\text{pred}$   
 $q)) = 1; \text{rsapss-sign } m\ e\ n \neq []; s = \text{rsapss-sign } m\ e\ n \rrbracket \implies \text{rsapss-verify } m\ s\ d$   
 $n = \text{True}$   
**apply** (*simp only: rsapss-sign rsapss-verify*)  
**apply** (*simp only: rsapss-sign-help1*)  
**apply** (*auto*)  
**apply** (*simp add: length-nat-to-bv-length*)  
**apply** (*simp add: bv-to-nat-nat-to-bv-length*)  
**apply** (*insert length-emsapss-encode [of m (length (nat-to-bv (p \* q)) - Suc 0)]*)  
**apply** (*insert bv-to-nat-emsapss-smaller-pq [of p q m]*)  
**apply** (*simp add: cryptinverts*)  
**apply** (*insert length-emsapss-encode [of m (length (nat-to-bv (p \* q)) - Suc 0)]*)  
**apply** (*insert nat-to-bv-length-bv-to-nat [of emsapss-encode m (length (nat-to-bv*  
 $(p * q)) - \text{Suc } 0) \text{ roundup } (\text{length } (\text{nat-to-bv } (p * q)) - \text{Suc } 0) \ 8 * 8]$ )  
**by** (*simp add: verify*)  
**end**

## References

- [1] R. S. Boyer and J. S. Moore. Proof checking the rsa public key encryption algorithm. Technical Report 33, Institute for Computing Science and Computer Applications, University of Texas, 1982.
- [2] P. Editor. PKCS#1 v2.1: RSA Cryptography Standard. Technical report, RSA Laboratories, 2002.
- [3] Development website of isabelle at the tu munich. <http://isabelle.in.tum.de>.
- [4] Mihir Bellare and Phillip Rogaway. PSS: Provably Secure Encoding Method for Digital Signatures. *Submission to IEEE P1363*, 1998.

- [5] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [6] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.
- [7] F. I. P. Standards. Secure hash standard. Technical Report FIPS 180-2, National Institute of Standards and Technology, 2002.