

RSAPSS

Christina Lindenberg and Kai Wirt
Darmstadt Technical University
Cryptography and Computeralgebra

June 10, 2026

Abstract

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

Contents

1 Extensions to the Word theory required for SHA1	2
2 Message Padding for SHA1	5
3 Formal definition of the secure hash algorithm	5
4 Definition of rsacrypt	9
5 Lemmata for modular arithmetic	9
6 Positive differences	10
7 Lemmata for modular arithmetic with primes	12
8 Correctness proof for RSA	12
9 Extensions to the Word theory required for PSS	14
10 EMSA-PSS encoding and decoding operation	21
11 RSS-PSS encoding and decoding operation	32

1 Extensions to the Word theory required for SHA1

```
theory WordOperations
imports Word
begin
```

```
type-synonym bv = bit list
```

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

```
definition
```

```
  bvxor: bvxor a b = bv-mapzip ( $\oplus_b$ ) a b
```

```
definition
```

```
  bvand: bvand a b = bv-mapzip ( $\wedge_b$ ) a b
```

```
definition
```

```
  bvor: bvor a b = bv-mapzip ( $\vee_b$ ) a b
```

```
primrec last where
```

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

```
primrec dellast where
```

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

```
fun bvrol where
```

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

```
fun bvrer where
```

```
  bvrer a 0 = a
| bvrer [] x = []
| bvrer x (Suc n) = bvrer (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: $\text{addmod32 } a \ b =$
 $\text{rev } (\text{select } (\text{rev } (\text{nat-to-bv } ((\text{bv-to-nat } a) + (\text{bv-to-nat } b)))) \ 0 \ 31)$

definition

bv-prepend: $\text{bv-prepend } x \ b \ \text{bv} = \text{replicate } x \ b \ @ \ \text{bv}$

primrec zerolist where

$\text{zerolist } 0 = []$
 $|\ \text{zerolist } (\text{Suc } n) = \text{zerolist } n \ @ \ [\text{Zero}]$

primrec hextobv where

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

primrec hexvtobv where

$\text{hexvtobv } [] = []$
 $|\ \text{hexvtobv } (x\#r) = \text{hextobv } x \ @ \ \text{hexvtobv } r$

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

proof (*induct i arbitrary: l*)

case 0

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

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 $\text{length } (\text{selecthelp } l \ (\text{Suc } x)) = (\text{Suc } x) + 1$

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:  $\bigwedge i. \forall l j. \exists k. \text{select } l \ i \ j = \text{select } k \ 0 \ (j - i)$ 
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  $\exists k. \text{select } l \ 0 \ j = \text{select } k \ 0 \ (j - (0::\text{nat}))$  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  $\exists k. \text{select } (tl \ l) \ x \ (j - (1::\text{nat})) = \text{select } k \ 0 \ (j - (1::\text{nat}) - (x::\text{nat}))$ 
  using Suc[of tl l j - (1::nat)] by auto
  ultimately have  $\exists k. \text{select } l \ (Suc \ x) \ j = \text{select } k \ 0 \ (j - (1::\text{nat}) - (x::\text{nat}))$ 
  using b by auto
  then show  $\exists k. \text{select } l \ (Suc \ x) \ j = \text{select } k \ 0 \ (j - Suc \ x)$  by simp
  qed
  qed
qed

lemma selectlenhelp3:  $\forall j. \text{select } l \ 0 \ j = \text{selecthelp } l \ j$ 
proof
  fix j
  show  $\text{select } l \ 0 \ j = \text{selecthelp } l \ j$ 
  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  $\exists k. select\ l\ i\ j = select\ k\ 0\ (j-i)$  by simp
then have  $\exists k. length\ (select\ l\ i\ j) = length\ (select\ k\ 0\ (j-i))$  by auto
then have  $c: \exists k. length\ (select\ l\ i\ j) = length\ (selecthelp\ k\ (j-i))$ 
  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: $zerocount\ n = (((n + 64) \text{ div } 512) + 1) * 512 - n - (65::nat)$

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* ($x\#\#r$) $n = (if\ n <= 0\ then\ x\#\#r\ else\ delblockhelp\ r\ (n-(1::nat)))$

definition *delblock* **where**

delblock: $delblock\ x = delblockhelp\ x\ 512$

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 (bvxor(bvxor(
  select x (32*(j-(3::nat))) (31+(32*(j-(3::nat)))))) (select x (32*(j-(8::nat)))
(31+(32*(j-(8::nat)))))) (bvxor(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 = addmod32 D (select H 96 127);
  EE = addmod32 E (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 :: bit list. length (delblockhelp xs n) <= length xs
proof -
  fix n and xs :: bit list
  show length (delblockhelp xs n)  $\leq$  length xs
  by (induct n rule: delblockhelp.induct) auto
qed
have  $\bigwedge x$  xs :: bit list. length (delblock (x#xs)) < Suc (length xs)
proof -
  fix x and xs :: bit list
  from aux have length (delblockhelp xs 511) < Suc (length xs)
  using le-less-trans [of length (delblockhelp xs 511) length xs] by auto
  then show length (delblock (x#xs)) < Suc (length xs) by (simp add: delblock)
qed
then show ?thesis
  by (relation measure ( $\lambda(b, x, A, B, C, D, E).$  length x)) auto
qed

```

definition IV1 **where**

```
IV1: IV1 = hexvtobv [x6,x7,x4,x5,x2,x3,x0,x1]
```

definition IV2 **where**

```
IV2: IV2 = hexvtobv [xE,xF,xC,xD,xA,xB,x8,x9]
```

definition IV3 **where**

```
IV3: IV3 = hexvtobv [x9,x8,xB,xA,xD,xC,xF,xE]
```

definition IV4 **where**

```
IV4: IV4 = hexvtobv [x1,x0,x3,x2,x5,x4,x7,x6]
```

definition IV5 **where**

```
IV5: IV5 = hexvtobv [xC,x3,xD,x2,xE,x1,xF,x0]
```

definition sha1 **where**

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

lemma sha1blocklen: 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: length (sha1 m) = 160

unfolding sha1 *Let-def* sha1blocklen ..

end

4 Definition of rsacrypt

```
theory Crypt
imports Main 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 rsa-crypt :: nat ⇒ nat ⇒ nat ⇒ nat
```

```
where
```

```
  cryptcorrect: rsa-crypt M e n = M ^ e mod n
```

```
lemma rsa-crypt-code [code]:
```

```
  rsa-crypt M e n = (if e = 0 then 1 mod n
    else if even e then rsa-crypt M (e div 2) n ^ 2 mod n
    else (M * rsa-crypt M (e div 2) n ^ 2 mod n) mod n)
```

```
proof -
```

```
  { fix m
```

```
    have (M ^ m mod n)2 mod n = (M ^ m)2 mod n
```

```
      by (simp add: power-mod)
```

```
    then have (M mod n) * ((M ^ m mod n)2 mod n) = (M mod n) * ((M ^ m)2 mod n)
```

```
      by simp
```

```
    have M * (M ^ m mod n)2 mod n = M * (M ^ m)2 mod n
```

```
      by (metis mod-mult-right-eq power-mod)
```

```
  }
```

```
  then show ?thesis
```

```
    by (auto simp add: cryptcorrect power-even-eq remainderexp elim!: evenE oddE)
```

```
qed
```

```
end
```

5 Leammata for modular arithmetic

```
theory Mod
imports Main
begin
```

```
lemma divmultassoc: a div (b*c) * (b*c) = ((a div (b * c)) * b)*(c::nat)
```

```
  by (rule mult.assoc [symmetric])
```

```
lemma delmod: (a::nat) mod (b*c) mod c = a mod c
```

```
  by (rule mod-mod-cancel [OF dvd-triv-right])
```

```
lemma timesmod1: ((x::nat)*(y::nat) mod n) mod (n::nat) = ((x*y) mod n)
```

```
  by (rule mod-mult-right-eq)
```

```
lemma timesmod3: ((a mod (n::nat)) * b) mod n = (a*b) mod n
```

```
  by (rule mod-mult-left-eq)
```

lemma *remainderexplemma*: $(y \text{ mod } (a::\text{nat}) = z \text{ mod } a) \implies (x*y) \text{ mod } a = (x*z) \text{ mod } a$
by (*rule mod-mult-cong [OF refl]*)

lemma *remainderexp*: $((a \text{ mod } (n::\text{nat}))^i) \text{ mod } n = (a^i) \text{ mod } n$
by (*rule power-mod*)

end

6 Positive differences

theory *Pdifference*
imports *HOL-Computational-Algebra.Primes Mod*
begin

definition

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

lemma *timesdistributesoverpdifference*:

$m*(\text{pdifference } a\ b) = \text{pdifference } (m*(a::\text{nat}))\ (m*(b::\text{nat}))$
by (*auto simp add: nat-distrib*)

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

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

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

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

lemma *equalmodstrick1*: $\text{pdifference } a\ b \text{ mod } p = 0 \implies a \text{ mod } p = b \text{ mod } p$
using *mod-eq-dvd-iff-nat [of a b p] mod-eq-dvd-iff-nat [of b a p]*
by (*cases a < b*) *auto*

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

lemma *diff-add-assoc2*: $a \leq c \implies c - (c - a + b) = (c - c + (a::\text{nat})) - b$
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:
assumes  $a \bmod p = b \bmod p$ 
shows pdifference  $a \bmod p = 0$ 
proof -
  { fix  $a \ b$ 
    assume  $*$ :  $a \bmod p = b \bmod p$ 
    have  $a - b = a \operatorname{div} p * p + a \bmod p - b \operatorname{div} p * p - b \bmod p$ 
      by simp
    also have  $\dots = a \operatorname{div} p * p - b \operatorname{div} p * p$ 
      using  $*$  by (simp only:)
    also have  $\dots = (a \operatorname{div} p - b \operatorname{div} p) * p$ 
      by (simp add: diff-mult-distrib)
    finally have  $(a - b) \bmod p = 0$ 
      by simp
  }
from this [OF assms] this [OF assms] [symmetric]
show ?thesis by simp
qed

lemma primekeyrewrite:
fixes  $p::nat$  shows  $\llbracket \text{prime } p; p \operatorname{dvd} (a*b); \sim(p \operatorname{dvd} a) \rrbracket \implies p \operatorname{dvd} b$ 
apply (subst (asm) prime-dvd-mult-nat)
apply auto
done

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

lemma primekeytrick:
fixes  $A \ B :: nat$ 
assumes  $(M * A) \bmod P = (M * B) \bmod P$ 
assumes  $M \bmod P \neq 0$  and prime  $P$ 
shows  $A \bmod P = B \bmod P$ 
proof -
from assms have  $M > 0$ 
by (auto intro: ccontr)
from assms have  $*$ :  $\bigwedge q. P \operatorname{dvd} M * q \implies P \operatorname{dvd} q$ 
using primekeyrewrite [of P M] unfolding dvd-eq-mod-eq-0 [symmetric] by
blast
from equalmodstrick2 [OF assms(1)]  $\langle M > 0 \rangle$  show ?thesis
apply -
apply (rule equalmodstrick1)
apply (auto intro: * dvdI simp add: dvd-eq-mod-eq-0 [symmetric] diff-mult-distrib2)

```

```

[symmetric)
  done
qed

end

```

7 Lemmata for modular arithmetic with primes

```

theory Productdivides
imports Pdifference
begin

```

```

lemma productdivides:  $\llbracket x \bmod a = (0::nat); x \bmod b = 0; \text{prime } a; \text{prime } b; a \neq b \rrbracket \implies x \bmod (a*b) = 0$ 
  by (simp add: mod-eq-0-iff-dvd primes-coprime divides-mult)

```

```

lemma specializedtoprimes1:
  fixes p::nat
  shows  $\llbracket \text{prime } p; \text{prime } q; p \neq q; a \bmod p = b \bmod p; a \bmod q = b \bmod q \rrbracket$ 
     $\implies a \bmod (p*q) = b \bmod (p*q)$ 
  by (metis equalmodstrick1 equalmodstrick2 productdivides)

```

```

lemma specializedtoprimes1a:
  fixes p::nat
  shows  $\llbracket \text{prime } p; \text{prime } q; p \neq q; a \bmod p = b \bmod p; a \bmod q = b \bmod q; b < p*q \rrbracket$ 
     $\implies a \bmod (p*q) = b$ 
  by (simp add: specializedtoprimes1)

```

```

end

```

8 Correctness proof for RSA

```

theory Cryptinverts
imports Crypt Productdivides HOL-Number-Theory.Residues
begin

```

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

```

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

```

```

lemma pred-unfold:
  pred n = n - 1
  by (induct n simp-all)

```

```

lemma fermat:

```

```

assumes prime p m mod p ≠ 0
shows m^(p-(1::nat)) mod p = 1
proof -
  from assms have [m^(p-1) = 1] (mod p)
    using fermat-theorem [of p m] by (simp add: mod-eq-0-iff-dvd)
  then show ?thesis
    using ⟨prime p⟩ prime-gt-1-nat [of p] by (simp add: cong-def)
qed

lemma cryptinverts-hilf1: prime p ⇒ (m * m^(k * pred p)) mod p = m mod p
apply (cases m mod p = 0)
apply (simp add: mod-mult-left-eq)
apply (simp only: mult.commute [of k pred p]
  power-mult mod-mult-right-eq [of m (m^pred p)^k p]
  remainderexp [of m^pred p p k, symmetric])
apply (insert fermat [of p m], auto)
apply (simp add: mult.commute [of k] power-mult pred-unfold)
by (metis One-nat-def mod-mult-right-eq mult.right-neutral power-Suc-0 power-mod)

lemma cryptinverts-hilf2: prime p ⇒ m*(m^(k * (pred p) * (pred q))) mod p =
m mod p
apply (simp add: mult.commute [of k * pred p pred q] mult.assoc [symmetric])
apply (rule cryptinverts-hilf1 [of p m (pred q) * k])
apply simp
done

lemma cryptinverts-hilf3: prime q ⇒ m*(m^(k * (pred p) * (pred q))) mod q =
m mod q
by (fact cryptinverts-hilf1)

lemma cryptinverts-hilf4:
  m^x mod (p * q) = m if prime p prime q p ≠ q
  m < p * q x mod (pred p * pred q) = 1
proof (cases x)
  case 0
    with that show ?thesis
      by simp
  next
    case (Suc x)
      with that(5) have Suc x mod (pred p * pred q) = Suc 0
        by simp
      then have pred p * pred q dvd x
        using dvd-minus-mod [of (pred p * pred q) Suc x]
        by simp
      then obtain y where x = pred p * pred q * y ..
      then have m^Suc x mod p = m mod p and m^Suc x mod q = m mod q
        using cryptinverts-hilf2 [of p m y q, OF ⟨prime p⟩]
          cryptinverts-hilf3 [of q m y p, OF ⟨prime q⟩]
        by (simp-all add: ac-simps)

```

```

with that Suc show ?thesis
  by (auto intro: specializedtoprimes1a)
qed

```

```

lemma primmultgreater: fixes p::nat shows [[ prime p; prime q; p ≠ 2; q ≠ 2]]
⇒ 2 < p*q
  apply (simp add: prime-nat-iff)
  apply (insert mult-le-mono [of 2 p 2 q])
  apply auto
  done

```

```

lemma primmultgreater2: fixes p::nat shows [[prime p; prime q; p ≠ q]] ⇒ 2
< p*q
  apply (cases p = 2)
  apply simp+
  apply (simp add: prime-nat-iff)
  apply (cases q = 2)
  apply (simp add: prime-nat-iff)
  apply (erule primmultgreater)
  apply auto
  done

```

```

lemma cryptinverts: [[ prime p; prime q; p ≠ q; n = p*q; m < n;
  e*d mod ((pred p)*(pred q)) = 1]] ⇒ rsa-crypt (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-nat-iff)
  apply (simp add: cryptcorrect remainderexp [of m ^e p*q d] power-mult [symmetric])
  done

```

```
end
```

9 Extensions to the Word theory required for PSS

```

theory Wordarith
imports WordOperations HOL-Computational-Algebra.Primes
begin

```

```
definition
```

```
  nat-to-bv-length :: nat ⇒ nat ⇒ bv where
```

```

  nat-to-bv-length:
  nat-to-bv-length n l = (if length(nat-to-bv n) ≤ l then bv-extend l 0 (nat-to-bv n)
else [])

```

```
lemma length-nat-to-bv-length:
```

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

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

definition

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

lemma rndvd : $b \ \text{dvd} \ a \implies \text{roundup } a \ b * b = a$
by ($\text{auto simp add: roundup dvd-eq-mod-eq-0}$)

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

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

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

proof ($\text{induct } a$)

show $\text{bv-to-nat } [] = \text{bv-to-nat } (\text{remzero } [])$
by simp

next

case ($\text{Cons } a1 \ a2$)

show $\text{bv-to-nat } a2 = \text{bv-to-nat } (\text{remzero } a2) \implies$
 $\text{bv-to-nat } (a1 \# a2) = \text{bv-to-nat } (\text{remzero } (a1 \# a2))$

proof ($\text{cases } a1$)

assume $a: a1 = \mathbf{0}$ **then have** $\text{bv-to-nat } (a1\#a2) = \text{bv-to-nat } a2$

using $\text{bv-to-nat-zero-prepend}$ **by** simp

moreover have $\text{remzero } (a1 \# a2) = \text{remzero } a2$ **using** a **by** simp

ultimately show $?thesis$ **using** Cons **by** simp

next

assume $a1 = \mathbf{1}$ **then show** $?thesis$ **by** simp

qed

qed

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

proof (auto)

assume $b: \text{nat-to-bv } a = []$

have $a: \text{bv-to-nat } [] = 0$ **by** simp

have $c: \text{bv-to-nat } (\text{nat-to-bv } a) = 0$ **using** a **and** b **by** simp

from x **have** $d: \text{bv-to-nat } (\text{nat-to-bv } a) = a$ **by** simp

from d **and** c **have** $a = 0$ **by** simp

then show False **using** x **by** simp

qed

lemma *remzero-replicate*: $\text{remzero } ((\text{replicate } n \ \mathbf{0})@l) = \text{remzero } l$
by (*induct n, auto*)

lemma *length-bv xor-bound*: $a \leq \text{length } l \implies a \leq \text{length } (\text{bv xor } l \ l2)$

proof (*induct a*)

case 0

then show ?*case* **by** *simp*

next

case (*Suc a*)

have $a: \text{Suc } a \leq \text{length } l$ **by** *fact*

with *Suc.hyps* **have** $b: a \leq \text{length } (\text{bv xor } l \ l2)$ **by** *simp*

show $\text{Suc } a \leq \text{length } (\text{bv xor } l \ l2)$

proof *cases*

assume $c: a = \text{length } (\text{bv xor } l \ l2)$

show $\text{Suc } a \leq \text{length } (\text{bv xor } l \ l2)$

proof (*simp add: bv xor*)

have $\text{length } l \leq \max (\text{length } l) (\text{length } l2)$ **by** *simp*

then show $\text{Suc } a \leq \max (\text{length } l) (\text{length } l2)$ **using** a **by** *simp*

qed

next

assume $a \neq \text{length } (\text{bv xor } l \ l2)$

then have $a < \text{length } (\text{bv xor } l \ l2)$ **using** b **by** *simp*

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

qed

qed

lemma *nat-to-bv-helper-legacy-induct*:

$(\bigwedge n. n \neq (0::\text{nat}) \longrightarrow P (n \text{ div } 2) \implies P n) \implies P x$

unfolding *atomize-imp[symmetric]*

by (*induction-schema, simp, lexicographic-order*)

lemma *len-lower-bound*:

assumes $0 < n$

shows $2^{\sim}(\text{length } (\text{nat-to-bv } n) - \text{Suc } 0) \leq n$

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-legacy-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 <= 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  $(2::\text{nat})^{\wedge}(\text{length}(\text{nat-to-bv-helper } (n \text{ div } 2) \ []) - \text{Suc } 0) * 2 =$ 
 $2^{\wedge}(\text{length}(\text{nat-to-bv-helper } (n \text{ div } 2) \ []) - \text{Suc } 0 + 1)$  by simp
  moreover have  $(2::\text{nat})^{\wedge}(\text{length}(\text{nat-to-bv-helper } (n \text{ div } 2) \ []) - \text{Suc } 0 +$ 
1) =
 $2^{\wedge}(\text{length}(\text{nat-to-bv-helper } (n \text{ div } 2) \ []))$ 
proof -
  have  $0 < n \text{ div } 2$  using  $a$  by arith
  then have  $0 < \text{length}(\text{nat-to-bv } (n \text{ div } 2))$  by (simp add: nat-to-bv-non0)
  then have  $0 < \text{length}(\text{nat-to-bv-helper } (n \text{ div } 2) \ [])$  using  $a$  by (simp add:
nat-to-bv-def)
  then show ?thesis by simp
qed
ultimately show  $(2::\text{nat})^{\wedge} \text{length}(\text{nat-to-bv-helper } (n \text{ div } 2) \ []) =$ 

```

$2^{\wedge}(\text{length}(\text{nat-to-bv-helper}(n \text{ div } 2) \ [])) - \text{Suc } 0) * 2$ **by simp**
qed
ultimately show $2^{\wedge}(\text{length}(\text{nat-to-bv-helper } n \ [])) - \text{Suc } 0) \leq n$
using b **by** (*simp add: nat-to-bv-def*) *arith*
qed
next
assume $c: \sim 1 < n$
show *?thesis*
proof (*cases n=1*)
assume $a: n=1$ **then have** $\text{nat-to-bv } n = [1]$ **by** (*simp add: nat-to-bv-non0*)
then show $2^{\wedge}(\text{length}(\text{nat-to-bv } n) - \text{Suc } 0) \leq n$ **using** a **by simp**
next
assume $n \sim 1$
with $\langle 0 < n \rangle$ **show** $2^{\wedge}(\text{length}(\text{nat-to-bv } n) - \text{Suc } 0) \leq n$ **using** c **by simp**
qed
qed

lemma *length-lower*: **assumes** $a: \text{length } a < \text{length } b$ **and** $b: (\text{hd } b) \sim 0$ **shows** $\text{bv-to-nat } a < \text{bv-to-nat } b$

proof –
have $ha: \text{bv-to-nat } a < 2^{\wedge} \text{length } a$ **by** (*simp add: bv-to-nat-upper-range*)
have $b \sim []$ **using** a **by auto**
then have $b = (\text{hd } b) \# (\text{tl } b)$ **by simp**
then have $\text{bv-to-nat } b = \text{bitval } (\text{hd } b) * 2^{\wedge}(\text{length } (\text{tl } b)) + \text{bv-to-nat } (\text{tl } b)$ **using**
bv-to-nat-helper[*of hd b tl b*] **by simp**
moreover have $\text{bitval } (\text{hd } b) = 1$
proof (*cases hd b*)
assume $hd\ b = 0$
then show $\text{bitval } (\text{hd } b) = 1$ **using** b **by simp**
next
assume $hd\ b = 1$
then show $\text{bitval } (\text{hd } b) = 1$ **by simp**
qed
ultimately have $hb: 2^{\wedge} \text{length } (\text{tl } b) \leq \text{bv-to-nat } b$ **by simp**
have $2^{\wedge}(\text{length } a) \leq (2::\text{nat})^{\wedge} \text{length } (\text{tl } b)$ **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** $\exists x. \text{nat-to-bv } n = x @ [\text{if } n \text{ mod } 2 = 0 \text{ then } 0 \text{ else } 1]$ **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*: $0 < n \implies \text{hd } (\text{nat-to-bv-helper } n \ []) = 1$

```

proof (induct n rule: nat-to-bv-helper-legacy-induct)
  fix n
  assume *:  $n \neq 0 \longrightarrow 0 < n \text{ div } 2 \longrightarrow \text{hd} (\text{nat-to-bv-helper } (n \text{ div } 2) []) = \mathbf{1}$ 
    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:  $0 < n \text{ div } 2 \longrightarrow \text{hd} (\text{nat-to-bv-helper } (n \text{ div } 2) []) = \mathbf{1}$  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  $\text{hd} (\text{nat-to-bv-helper } n []) =$ 
       $\text{hd} (\text{nat-to-bv } (n \text{ div } 2) @ [\text{if } n \text{ mod } 2 = 0 \text{ then } \mathbf{0} \text{ else } \mathbf{1}])$ 
      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
       $\text{nat-to-bv-non-empty}[of n \text{ div } 2]$  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:
  fixes p::nat assumes a: prime p and b: prime q shows  $\text{hd} (\text{nat-to-bv } (p*q)) \sim =$ 
  0
proof –
  have  $0 < p*q$ 
    by (metis a b mult-is-0 neq0-conv not-prime-0)
  then show ?thesis using hd-one[of p*q] and nat-to-bv-def by auto
qed

```

```

lemma primerev: fixes p::nat shows  $[[m \text{ dvd } p; m \sim = 1; m \sim = p]] \implies \sim \text{prime } p$ 
by (auto simp add: prime-nat-iff)

```

```

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

```

```

lemma exp-prod1:  $[[1 < b; 2^x = 2*(b::nat)]] \implies 2 \text{ dvd } b$ 
proof –
  assume a:  $1 < b$  and b:  $2^x = 2*(b::nat)$ 
  have s1:  $1 < x$ 

```

```

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

lemma exp-prod2: [1 < a; 2^x = a*2] ==> (2::nat) dvd a
proof -
  assume 2^x = a*2
  then have 2^x = 2*a by simp
  moreover assume 1 < a
  ultimately show 2 dvd a using exp-prod1 by simp
qed

lemma odd-mul-odd: [~(2::nat) dvd p; ~2 dvd q] ==> ~2 dvd p*q
  by simp

lemma prime-equal: fixes p::nat shows [prime p; prime q; 2^x = p*q] ==> (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-nat-iff)
  moreover from b have e: 1 < q by (simp add: prime-nat-iff)
  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~=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 primerew[of 2 p] and a by auto
then show ?thesis using p by simp
next
assume q: q~=2 show p=q
proof -
  from p have ~ 2 dvd p using primerew and a by auto
  moreover from q have ~2 dvd q using primerew and b by auto
  ultimately have ~2 dvd p*q by (simp add: odd-mul-odd)
  then have odd ((2 :: nat) ^ x) by (simp only: c) simp
  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~=0 then have 0<x by simp
    then show ?thesis using two-dvd-exp by simp
  qed
  ultimately show ?thesis by simp
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

10 EMSA-PSS encoding and decoding operation

```

theory EMSAPSS
imports SHA1 Wordarith
begin

```

We define the encoding and decoding operations for the probabilistic signature scheme. Finally we show, that encoded messages always can be verified

```

definition show-rightmost-bits:: bv  $\Rightarrow$  nat  $\Rightarrow$  bv
  where show-rightmost-bits bvec n = rev (take n (rev bvec))

```

```

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

```

```

definition salt:: bv
  where salt = []

```

definition *sLen*:: *nat*
where *sLen* = *length salt*

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

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

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

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

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

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

definition *generate-salt*:: *bv* \Rightarrow *bv*
where *generate-salt DB-zero* = *show-rightmost-bits DB-zero sLen*

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

definition *MGF1*:: *bv* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *bv*
where *MGF1 Z n l* = *take l (MGF2 Z n)*

definition *MGF*:: *bv* \Rightarrow *nat* \Rightarrow *bv*
where
MGF Z l = *(if l = 0 \vee 2³²*(length (sha1 Z)) < l*
then []
else MGF1 Z (roundup l (length (sha1 Z)) - 1) l)

definition *emsapss-encode-help8*:: *bv* \Rightarrow *bv* \Rightarrow *bv*
where *emsapss-encode-help8 DBzero H* = *DBzero @ H @ BC*

definition *emsapss-encode-help7*:: *bv* \Rightarrow *bv* \Rightarrow *nat* \Rightarrow *bv*
where *emsapss-encode-help7 maskedDB H emBits* =

emsapss-encode-help8 (*maskedDB-zero maskedDB emBits*) *H*

definition *emsapss-encode-help6*:: *bv* \Rightarrow *bv* \Rightarrow *bv* \Rightarrow *nat* \Rightarrow *bv*
where *emsapss-encode-help6* *DB dbMask H emBits* =
 (*if dbMask* = []
 then []
 else emsapss-encode-help7 (*bxor DB dbMask*) *H emBits*)

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

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

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

definition *emsapss-encode-help2*:: *bv* \Rightarrow *nat* \Rightarrow *bv*
where *emsapss-encode-help2* *M' emBits* = *emsapss-encode-help3* (*sha1 M'*) *emBits*

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

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

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

definition *emsapss-decode-help10*:: *bv* \Rightarrow *bv* \Rightarrow *bool*
where *emsapss-decode-help10* *M' H* = *emsapss-decode-help11* (*sha1 M'*) *H*

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

definition *emsapss-decode-help8*:: *bv* \Rightarrow *bv* \Rightarrow *bv* \Rightarrow *bool*
where *emsapss-decode-help8* *mHash DB-zero H* =

emsapss-decode-help9 *mHash* (*generate-salt DB-zero*) *H*

definition *emsapss-decode-help7*:: *bv* ⇒ *bv* ⇒ *bv* ⇒ *nat* ⇒ *bool*

where *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*)

definition *emsapss-decode-help6*:: *bv* ⇒ *bv* ⇒ *bv* ⇒ *nat* ⇒ *bool*

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

definition *emsapss-decode-help5*:: *bv* ⇒ *bv* ⇒ *bv* ⇒ *bv* ⇒ *nat* ⇒ *bool*

where *emsapss-decode-help5* *mHash* *maskedDB* *dbMask* *H* *emBits* =
emsapss-decode-help6 *mHash* (*bv**xor* *maskedDB* *dbMask*) *H* *emBits*

definition *emsapss-decode-help4*:: *bv* ⇒ *bv* ⇒ *bv* ⇒ *nat* ⇒ *bool*

where *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*)

definition *emsapss-decode-help3*:: *bv* ⇒ *bv* ⇒ *nat* ⇒ *bool*

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

definition *emsapss-decode-help2*:: *bv* ⇒ *bv* ⇒ *nat* ⇒ *bool*

where *emsapss-decode-help2* *mHash* *EM* *emBits* =
 (if *show-rightmost-bits* *EM* 8 ≠ *BC*
 then *False*
 else *emsapss-decode-help3* *mHash* *EM* *emBits*)

definition *emsapss-decode-help1*:: *bv* ⇒ *bv* ⇒ *nat* ⇒ *bool*

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

definition *emsapss-decode*:: *bv* ⇒ *bv* ⇒ *nat* ⇒ *bool*

where *emsapss-decode* *M* *EM* *emBits* =
 (if ($2^{64} \leq$ length *M* ∨ $2^{32} * 160 <$ *emBits*)
 then *False*
 else *emsapss-decode-help1* (*sha1* *M*) *EM* *emBits*)

lemma *roundup-positiv*: $0 < emBits \implies 0 < roundup\ emBits\ 160$
by (*auto simp add: roundup*)

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 mult-div-mod-eq [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*: $0 < emBits \implies roundup\ emBits\ 8 * 8 - emBits \leq 7$
by (*auto simp add: roundup arith*)

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-def*)
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*: $\llbracket 8 \leq roundup\ emBits\ 8 * 8 - 168 \rrbracket \implies 176 \leq roundup\ emBits\ 8 * 8$
by *arith*

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

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*: $\llbracket 176 + sLen \leq emBits \rrbracket \implies \text{roundup } emBits \ 8 * 8 - emBits \leq \text{roundup } emBits \ 8 * 8 - 160 - sLen - 16$

by (*simp* *add: sLen-def*)

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

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

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

apply (*insert* *bv-prepend-append2* [of *x y b []*])

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

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

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

lemma *length-generate-PS*: $\text{length (generate-PS emBits 160)} = (\text{roundup emBits } 8) * 8 - sLen - 160 - 16$
by (*simp add: generate-PS-def length-bv-prepend*)

lemma *length-bvxor*: $\text{length } a = \text{length } b \implies \text{length (bxor } a \text{ b)} = \text{length } a$
by (*simp add: bxor*)

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-def 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-def 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*:
 $\llbracket 0 < \text{emBits}; \text{length (sha1 } M) + sLen + 16 \leq \text{emBits} \rrbracket$

$\implies \text{length } (\text{generate-DB } (\text{generate-PS } \text{emBits } (\text{length } (\text{sha1 } x)))) = (\text{roundup } \text{emBits } 8) * 8 - 168$
apply (*insert roundup-ge-emBits [of emBits 8]*)
apply (*simp add: length-generate-DB length-generate-PS sha1len*)
done

lemma *length-maskedDB-zero:*

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

lemma *take-equal-bv-prepend:*

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

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

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

lemma *equal-zero:*

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

lemma *get-salt: $\llbracket 176 + \text{sLen} \leq \text{emBits}; \text{roundup } \text{emBits } 8 * 8 - \text{emBits} \leq 7 \rrbracket \implies$*
(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-def generate-PS-def maskedDB-zero-def*)
apply (*simp add: bv-prepend-split bv-prepend-drop generate-salt-def*
show-rightmost-bits-def sLen-def)
done

lemma *generate-maskedDB-elim: $\llbracket \text{roundup } \text{emBits } 8 * 8 - \text{emBits} \leq \text{length } x;$*
*(roundup emBits 8) * 8 - (length (sha1 M)) - 8 = length (maskedDB-zero*
x emBits) $\rrbracket \implies \text{generate-maskedDB } (\text{maskedDB-zero } x \text{emBits } @ y @ z) \text{emBits}$
(length (sha1 M)) = maskedDB-zero x emBits

apply (*simp add: maskedDB-zero-def*)
apply (*insert length-bv-prepend-drop [of (roundup emBits 8 * 8 - emBits) x]*)
apply (*simp add: generate-maskedDB-def*)
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-def*)
apply (*insert length-bv-prepend-drop [of roundup emBits 8 * 8 - emBits x]*)
apply (*simp add: generate-H-def*)
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*: $\llbracket 176 + sLen \leq emBits; \text{roundup } emBits \ 8 * 8 - emBits \leq 7 \rrbracket \implies \text{take } 8 \ (\text{drop } (\text{roundup } emBits \ 8 * 8 - (\text{length } (\text{sha1 } M) + sLen + 16)) \ (\text{maskedDB-zero } (\text{generate-DB } (\text{generate-PS } emBits \ 160)) \ emBits)) = [\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{1}]$
apply (*insert roundup-help2 [of emBits] length-generate-PS [of emBits] equal-zero [of emBits]*)
apply (*simp add: sha1len maskedDB-zero-def generate-DB-def generate-PS-def 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 $\text{bvxor } (\text{bvxor } a \ b) \ b = a$
proof –
have $\bigwedge b. \ \text{length } a = \text{length } b \implies \text{bvxor } (\text{bvxor } a \ b) \ b = a$
apply (*induct a*)
apply (*auto simp add: bvxor*)
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 *bvxorxor-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 (⊕b) a b))) = length
    b)
      apply (subgoal-tac  $b = (\text{take } x \ b) @ (\text{drop } x \ b)$ )
      apply (insert drop-bv-mapzip [of x (replicate x 0 @ drop x (bv-mapzip (⊕b) a
    b)) b (⊕b)])
      apply (simp)
      apply (insert drop-bv-mapzip [of x a b (⊕b)])
      apply (simp)
      apply (fold bxor)
      apply (simp-all)
      done
  with assms show ?thesis by simp
qed

```

lemma *bvxorxor-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-def bxorxor-elim-help)

lemma *verify*: $\llbracket (\text{emsapss-encode } M \ \text{emBits}) \neq []; EM = (\text{emsapss-encode } M \ \text{emBits}) \rrbracket \implies \text{emsapss-decode } M \ EM \ \text{emBits} = \text{True}$
 apply (simp add: emsapss-decode-def emsapss-encode-def)
 apply (safe, simp+)
 apply (simp add: emsapss-decode-help1-def emsapss-encode-help1-def)
 apply (safe, simp+)
 apply (simp add: emsapss-decode-help2-def emsapss-encode-help2-def)
 apply (safe)

```

apply (simp add: emsapss-encode-help3-def emsapss-encode-help4-def
  emsapss-encode-help5-def emsapss-encode-help6-def)
apply (safe)
apply (simp add: emsapss-encode-help7-def emsapss-encode-help8-def lastbits-BC
  [symmetric])+
apply (simp add: emsapss-decode-help3-def emsapss-encode-help3-def
  emsapss-decode-help4-def emsapss-encode-help4-def)
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-def emsapss-encode-help5-def emsapss-encode-help6-def)
apply (safe)
apply (simp)
apply (simp add: emsapss-encode-help7-def)
apply (simp only: emsapss-encode-help8-def)
apply (simp only: maskedDB-zero-def)
apply (simp only: take-bv-prepend2 min.absorb1)
apply (simp)
apply (simp add: emsapss-encode-help5-def emsapss-encode-help6-def)
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-def emsapss-encode-help8-def)
apply (insert length-bvxor [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 (bvxor (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 (bvxor (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 (bvxor (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-def)
apply (simp only: emsapss-decode-help6-def emsapss-decode-help7-def)
apply (insert bvrxor-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-def)
apply (insert take-equal-bv-prepend [of emBits M] x01-elim [of emBits M] get-salt
  [of emBits])
apply (simp add: emsapss-decode-help8-def emsapss-decode-help9-def emsapss-decode-help10-def
  emsapss-decode-help11-def)
done

```

end

11 RSS-PSS encoding and decoding operation

```
theory RSAPSS
imports EMSAPSS Cryptinverts
begin
```

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

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

```
definition rsapss-sign:: bv ⇒ nat ⇒ nat ⇒ bv
  where 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))
```

```
definition rsapss-verify:: bv ⇒ bv ⇒ nat ⇒ nat ⇒ bool
  where rsapss-verify m s d n =
    (if (length s) ≠ 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 ≠ [] ⇒ length (emsapss-encode m x) = roundup x 8 * 8
  apply (atomize (full))
  apply (simp add: emsapss-encode-def)
  apply (simp add: emsapss-encode-help1-def)
  apply (simp add: emsapss-encode-help2-def)
  apply (simp add: emsapss-encode-help3-def)
  apply (simp add: emsapss-encode-help4-def)
  apply (simp add: emsapss-encode-help5-def)
  apply (simp add: emsapss-encode-help6-def)
  apply (simp add: emsapss-encode-help7-def)
  apply (simp add: emsapss-encode-help8-def)
  apply (simp add: maskedDB-zero-def)
  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-def)
  apply (simp add: MGF1-def)
  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-def)
apply (insert roundup-ge-emBits [of x 8])
apply safe
apply (simp add: max.absorb1)
done

```

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

```

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*

```

(bv xor
 (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
 (bv xor
 (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: length BC = 8 by (simp add: BC-def)
have b: length (sha1 (generate-M' (sha1 m) 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*: $\text{MGF } z \ l \ \sim = [] \implies l \leq 2^{\wedge}32 * (\text{length } (\text{sha1 } z))$

```

proof (cases  $2^{\wedge}32 * \text{length } (\text{sha1 } z) < l$ )
assume x:  $\text{MGF } z \ l \ \sim = []$ 
assume a:  $2^{\wedge}32 * \text{length } (\text{sha1 } z) < l$ 
then have  $\text{MGF } z \ l = []$ 
proof (cases l=0)

```

```

    assume l=0
    then show MGF z l = [] by (simp add: MGF-def)
next
    assume l~=0
    then have (l = 0 ∨ 2^32*length(sha1 z) < l) = True using a by fast
    then show MGF z l = [] apply (simp only: MGF-def) by simp
qed
then show ?thesis using x by simp
next
    assume ¬ 2 ^ 32 * length (sha1 z) < l
    then show ?thesis by simp
qed

lemma length-helper2: assumes p: prime p and q: prime q
    and 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)))))) ~ = []
and len: length (sha1 M) + sLen + 16 ≤ (length (nat-to-bv (p * q)) - Suc 0)
shows length
(
(bv xor
(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
proof -
    have a: length (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
(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-def)
    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-nat-iff)
      moreover from q have 1 < q by (simp add: prime-nat-iff)
      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 mod 8 ~ = 0 ⇒ (roundup emBits 8) * 8
- (8 - (emBits mod 8)) = emBits
by (auto simp add: roundup)

lemma length-bound: [emBits mod 8 ~ = 0; 8 <= (length maskedDB)] ⇒ length
(remzero ((maskedDB-zero maskedDB emBits)@a@b)) <= length (maskedDB@a@b)
- (8 - (emBits mod 8))
proof -
  assume a: emBits mod 8 ~ = 0
  assume len: 8 <= (length maskedDB)
  have b: ∧ a. length (remzero a) <= length a

```

```

proof –
  fix a
  show length (remzero a) <= length a
  proof (induct a)
    show (length (remzero [])) <= length [] by (simp)
  next
    case (Cons hd tl)
    show (length (remzero (hd#tl))) <= 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
qed
from len show length (remzero (maskedDB-zero maskedDB emBits @ a @ b))
≤ length (maskedDB @ a @ b) – (8 – emBits mod 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 mod 8 using a by simp
  moreover have length ((drop (8–emBits mod 8) maskedDB)@a@b) = length
(maskedDB@a@b) – (8–emBits mod 8)
  proof –
    show ?thesis using length-drop[of (8–emBits mod 8) maskedDB]
    proof (simp)
      have 0 <= emBits mod 8 by simp
      then have 8–(emBits mod 8) <= 8 by simp
      then show length maskedDB + emBits mod 8 – 8 + (length a + length
b) =
        length maskedDB + (length a + length b) + emBits mod 8 – 8 using len
by arith
    qed
  qed
  ultimately show ?thesis using b[of (drop ((roundup emBits 8)*8 – emBits)
maskedDB)@a@b]
  by (simp add: maskedDB-zero-def)
qed
qed

lemma length-bound2: 8 <= length ( (bv xor
(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))))))))
proof -
  have 8 <= 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-def)
  then show ?thesis using length-bv xor-bound by simp
qed

lemma length-helper: assumes p: prime p and q: prime q and x: (length (nat-to-bv
(p * q)) - Suc 0) mod 8 ~ = 0 and 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)))))) ~ = []
and len: length (sha1 M) + sLen + 16 ≤ (length (nat-to-bv (p * q)) - Suc 0)
shows length
(remzero
(maskedDB-zero
(bv xor
(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))
proof -
  from mgf have round: 168 <= roundup (length (nat-to-bv (p * q)) - Suc 0) 8
* 8
  proof (simp only: sha1len sLen-def)
    from len have 160 + sLen + 16 ≤ length (nat-to-bv (p * q)) - Suc 0 by (simp
add: sha1len)
    then have len1: 176 <= length (nat-to-bv (p * q)) - Suc 0 by simp
    have length (nat-to-bv (p*q)) - Suc 0 <= (roundup (length (nat-to-bv (p * q))
- Suc 0) 8) * 8
  - unfolding roundup
    proof (cases (length (nat-to-bv (p*q)) - Suc 0) mod 8 = 0)
      assume len2: (length (nat-to-bv (p * q)) - Suc 0) mod 8 = 0

```

then have (if (length (nat-to-bv (p * q)) - Suc 0) mod 8 = 0 then (length (nat-to-bv (p * q)) - Suc 0) div 8 else (length (nat-to-bv (p * q)) - Suc 0) div 8 + 1) * 8 = (length (nat-to-bv (p * q)) - Suc 0) div 8 * 8 **by simp**

moreover have (length (nat-to-bv (p * q)) - Suc 0) div 8 * 8 = (length (nat-to-bv (p * q)) - Suc 0) **using len2**

by auto

ultimately show length (nat-to-bv (p * q)) - Suc 0

 \leq (if (length (nat-to-bv (p * q)) - Suc 0) mod 8 = 0 then (length (nat-to-bv (p * q)) - Suc 0) div 8 else (length (nat-to-bv (p * q)) - Suc 0) div 8 + 1) * 8

by simp

next

assume len2: (length (nat-to-bv (p*q)) - Suc 0) mod 8 \sim 0

then have (if (length (nat-to-bv (p * q)) - Suc 0) mod 8 = 0 then (length (nat-to-bv (p * q)) - Suc 0) div 8 else (length (nat-to-bv (p * q)) - Suc 0) div 8 + 1) * 8 = ((length (nat-to-bv (p * q)) - Suc 0) div 8 + 1) * 8 **by simp**

moreover have length (nat-to-bv (p*q)) - Suc 0 \leq ((length (nat-to-bv (p*q)) - Suc 0) div 8 + 1) * 8 **by auto**

ultimately show length (nat-to-bv (p * q)) - Suc 0

 \leq (if (length (nat-to-bv (p * q)) - Suc 0) mod 8 = 0 then (length (nat-to-bv (p * q)) - Suc 0) div 8 else (length (nat-to-bv (p * q)) - Suc 0) div 8 + 1) * 8

by simp

qed

then show 168 \leq roundup (length (nat-to-bv (p * q)) - Suc 0) 8 * 8 **using len1 by simp**

qed

from x have a: length

 (remzero

 (maskedDB-zero

 (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))))))

 (length (nat-to-bv (p * q)) - Suc 0) @

 sha1 (generate-M' (sha1 m) salt) @ BC) \leq 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) - (8 - (length (nat-to-bv (p*q)) - Suc 0) mod 8) **using length-bound and length-bound2 by simp**

have b : $\text{length (bvxor (generate-DB (generate-PS (length (nat-to-bv (p * q)) - Suc 0) (length (sha1 (generate-M' (sha1 m) salt)))))) (MGF (sha1 (generate-M' (sha1 m) salt)) (length (generate-DB (generate-PS (length (nat-to-bv (p * q)) - Suc 0) (length (sha1 (generate-M' (sha1 m) salt)))))) @ sha1 (generate-M' (sha1 m) salt) @ BC) = length (bvxor (generate-DB (generate-PS (length (nat-to-bv (p * q)) - Suc 0) (length (sha1 (generate-M' (sha1 m) salt)))))) (MGF (sha1 (generate-M' (sha1 m) salt)) (length (generate-DB (generate-PS (length (nat-to-bv (p * q)) - Suc 0) (length (sha1 (generate-M' (sha1 m) salt)))))) + 168 \text{ using } \text{length-helper1} \text{ by simp}$

have c : $\text{length (bvxor (generate-DB (generate-PS (length (nat-to-bv (p * q)) - Suc 0) (length (sha1 (generate-M' (sha1 m) salt)))))) (MGF (sha1 (generate-M' (sha1 m) salt)) (length (generate-DB (generate-PS (length (nat-to-bv (p * q)) - Suc 0) (length (sha1 (generate-M' (sha1 m) salt)))))) =$

$(\text{roundup (length (nat-to-bv (p * q)) - Suc 0) 8) * 8 - 168 \text{ using } p \text{ and } q \text{ and } \text{length-helper2} \text{ and } \text{mgf} \text{ and } \text{len} \text{ by simp}$

from a **and** b **and** c **have** $\text{length (remzero (maskedDB-zero (bvxor (generate-DB (generate-PS (length (nat-to-bv (p * q)) - Suc 0) (length (sha1 (generate-M' (sha1 m) salt)))))) (MGF (sha1 (generate-M' (sha1 m) salt)) (length (generate-DB (generate-PS (length (nat-to-bv (p * q)) - Suc 0) (length (sha1 (generate-M' (sha1 m) salt)))))) (length (nat-to-bv (p * q)) - Suc 0) @ sha1 (generate-M' (sha1 m) salt) @ BC)) <= roundup (length (nat-to-bv (p * q)) - Suc 0) 8 * 8 - 168 + 168 - (8 - (length (nat-to-bv (p * q)) - Suc 0) mod 8) \text{ by simp}$

then have $\text{length (remzero (maskedDB-zero (bvxor (generate-DB (generate-PS (length (nat-to-bv (p * q)) - Suc 0) (length (sha1 (generate-M' (sha1 m) salt)))))) (MGF (sha1 (generate-M' (sha1 m) salt)) (length (generate-DB (generate-PS (length (nat-to-bv (p * q)) - Suc 0) (length (sha1 (generate-M' (sha1 m) salt)))))) (length (nat-to-bv (p * q)) - Suc 0) @ sha1 (generate-M' (sha1 m) salt) @ BC)) <= roundup (length (nat-to-bv (p * q)) - Suc 0) 8 * 8 - (8 - (length (nat-to-bv (p * q)) - Suc 0) mod 8) \text{ using } \text{round} \text{ by simp}$

moreover have $\text{roundup (length (nat-to-bv (p * q)) - Suc 0) 8 * 8 - (8 - (length (nat-to-bv (p * q)) - Suc 0) mod 8) = (length (nat-to-bv (p * q)) - Suc 0) \text{ using } x \text{ and } \text{emBits-roundup-cancel2} \text{ by simp}$

moreover have $0 < \text{length (nat-to-bv (p * q))}$

proof –

from p **have** s : $1 < p$ **by** $(\text{simp add: prime-nat-iff})$

moreover from q **have** $1 < q$ **by** $(\text{simp add: prime-nat-iff})$

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

proof –

assume *a*: *emsapss-encode* *m* (length (nat-to-bv (p * q)) - Suc 0) $\neq \square$

assume *p*: *prime* *p* **and** *q*: *prime* *q*

assume *x*: (length (nat-to-bv (p * q)) - Suc 0) mod 8 $\sim = 0$

have *b*: *emsapss-encode* *m* (length (nat-to-bv (p * q)) - Suc 0) = *emsapss-encode-help1* (sha1 *m*)

(length (nat-to-bv (p * q)) - Suc 0)

proof (*simp only: emsapss-encode-def*)

from *a* **show** (if (($2^{64} \leq \text{length } m$) \vee ($2^{32} * 160 < \text{length (nat-to-bv (p*q)) - Suc 0}$)))

then \square

else (*emsapss-encode-help1* (sha1 *m*) (length (nat-to-bv (p*q)) - Suc 0)) = (*emsapss-encode-help1* (sha1 *m*) (length (nat-to-bv (p*q)) - Suc 0))

by (*auto simp add: emsapss-encode-def*)

qed

have *c*: length (remzero (*emsapss-encode-help1* (sha1 *m*) (length (nat-to-bv (p * q)) - Suc 0))) < length (nat-to-bv (p*q))

proof (*simp only: emsapss-encode-help1-def*)

from *a* **and** *b* **have** *d*: (if ((length (nat-to-bv (p * q)) - Suc 0) < (length (sha1 *m*) + sLen + 16)))

then \square

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))

by (*auto simp add: emsapss-encode-def emsapss-encode-help1-def*)

from *d* **have** *len*: length (sha1 *m*) + sLen + 16 \leq (length (nat-to-bv (p*q))) - Suc 0

proof (*cases* length (nat-to-bv (p * q)) - Suc 0 < length (sha1 *m*) + sLen + 16)

assume length (nat-to-bv (p * q)) - Suc 0 < length (sha1 *m*) + sLen + 16

then **have** *len1*: (if length (nat-to-bv (p * q)) - Suc 0 < length (sha1 *m*) + sLen + 16 then \square)

else *emsapss-encode-help2* (generate-*M'* (sha1 *m*) salt) (length (nat-to-bv (p * q)) - Suc 0) = \square **by** *simp*

assume *len2*: (if length (nat-to-bv (p * q)) - Suc 0 < length (sha1 *m*) + sLen + 16 then \square)

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 \leq length (nat-to-bv (p * q)) - Suc 0

```

    by (auto simp add: emsapss-encode-def emsapss-encode-help1-def)
  next
    assume  $\neg$  length (nat-to-bv (p * q)) - Suc 0 < length (sha1 m) + sLen +
16
    then show length (sha1 m) + sLen + 16  $\leq$  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-def)
      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-def emsapss-encode-help4-def em-
sapss-encode-help5-def)
        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-def)
            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))))))
                )  $\sim$  =
              [] by (auto simp add: emsapss-encode-def emsapss-encode-help1-def
emsapss-encode-help2-def emsapss-encode-help3-def emsapss-encode-help4-def em-
sapss-encode-help5-def emsapss-encode-help6-def)
            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)))
              (length (nat-to-bv (p * q)) - Suc 0)))
        (by (auto simp add: emsapss-encode-def emsapss-encode-help1-def
          emsapss-encode-help2-def
            emsapss-encode-help3-def emsapss-encode-help4-def emsapss-encode-help5-def
              emsapss-encode-help6-def)
          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-def emsapss-encode-help8-def)
              from p and q and x show length
                (remzero
                  (maskedDB-zero
                    (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)) - Suc 0)))
                    (length (nat-to-bv (p * q)) - Suc 0)))
                (length (nat-to-bv (p * q)) - Suc 0)))
            (length (nat-to-bv (p * q)) - Suc 0)))
          (length (nat-to-bv (p * q)) - Suc 0)))
      (length (nat-to-bv (p * q)) - Suc 0)))
    (length (nat-to-bv (p * q)) - Suc 0)))

```

```

      (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
              (bwrap
               (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** *pneq*: $p \neq q$ **and** *c*: *emsapss-encode m* ($\text{length (nat-to-bv (p * q)) - Suc 0} \neq []$) **shows** *bv-to-nat (emsapss-encode m (length (nat-to-bv (p * q)) - Suc 0))* < *p*q*
proof –
 from *a* **and** *b* **and** *c* **show** ?thesis

proof (*cases* 8 *dvd* ((*length* (*nat-to-bv* (*p * q*))) - *Suc 0*))
assume *d*: 8 *dvd* ((*length* (*nat-to-bv* (*p * q*))) - *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-nat-iff*)
moreover have $0 < q$ **using** *b* **by** (*simp add: prime-nat-iff*)
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** *pneq* **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** $\text{bv-to-nat}(\text{emsapss-encode } m(\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0)) < 2^{\wedge}(\text{roundup}(\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0) * 8)$
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*: $\text{bv-to-nat}(\text{emsapss-encode } m(\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0)) = \text{bv-to-nat}(\text{remzero}(\text{emsapss-encode } m(\text{length}(\text{nat-to-bv}(p * q)) - \text{Suc } 0)))$ **by** *simp*
from *a* **and** *b* **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(length(nat-to-bv(p * q)) - Suc 0)) 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-def rsapss-verify-def*)

apply (*simp only: rsapss-sign-help1-def*)

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)) - Suc 0) roundup (length (nat-to-bv (p * q)) - Suc 0) 8 * 8]*)

apply (*simp add: verify*)

done

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.