RSAPSS

Christina Lindenberg and Kai Wirt
Darmstadt Technical University
Cryptography and Computeralgebra

June 17, 2024

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

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 *bvror* **where**
  bvror a 0 = *a*
| *bvror* [] *x* = []
| *bvror x* (*Suc n*) = *bvror* (*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 hextobv where
  hextobv x0 = [Zero,Zero,Zero,Zero]
| hextobv x1 = [Zero,Zero,Zero,One]
| hextobv x2 = [Zero,Zero,One,Zero]
| hextobv x3 = [Zero,Zero,One,One]
| hextobv x4 = [Zero,One,Zero,Zero]
| hextobv x5 = [Zero,One,Zero,One]
| hextobv x6 = [Zero,One,One,Zero]
| hextobv x7 = [Zero,One,One,One]
| hextobv x8 = [One,Zero,Zero,Zero]
| hextobv x9 = [One,Zero,Zero,One]
| hextobv xA = [One,Zero,One,Zero]
| hextobv xB = [One,Zero,One,One]
| hextobv xC = [One,One,Zero,Zero]
| hextobv xD = [One,One,Zero,One]
| hextobv xE = [One,One,One,Zero]
| hextobv xF = [One,One,One,One]

primrec hexvtobv where
  hexvtobv [] = []
| hexvtobv (x#r) = hextobv x @ hexvtobv r

lemma selectlenhelp: length (selecthelp l i) = (i + 1)
proof (induct i arbitrary: l)
  case 0
  show length (selecthelp l 0) = 0 + 1
  proof (cases l)
    case Nil
    then have selecthelp l 0 = [Zero] by simp
    then show ?thesis by simp
  next
    case (Cons a as)
    then have 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: ⋀i. ∀ l j. ∃ k. select l i j = select k 0 (j − i)
proof (auto)
  fix i
  show ⋀l j. ∃ k. select l i j = select k 0 (j − i)
  proof (induct i)
    fix l and j
    have select l 0 j = select l 0 (j−(0::nat)) by simp
    then show ∃ k. select l 0 j = select k 0 (j − (0::nat)) by auto
  next
    case (Suc x)
    have b: select l (Suc x) j = select (tl l) x (j−(1::nat))
    proof (cases l)
      case Nil
      then have select l (Suc x) j = select l x (j−(1::nat)) by simp
      moreover have tl l = l using Nil by simp
      ultimately show ?thesis by (simp)
    next
      case (Cons head tail)
      then have select l (Suc x) j = select tail x (j−(1::nat)) by simp
      moreover have tail = tl l using Cons by simp
      ultimately show ?thesis by simp
    qed
    have ∃ k. select l x j = select k 0 (j − (x::nat)) using Suc by simp
   moreover have ∃ k. select (tl l) x (j−(1::nat)) = select k 0 (j−(1::nat)−(x::nat))
using Suc[of tl l j−(1::nat)] by auto
     ultimately have ∃ k. select l (Suc x) j = select k 0 (j−(1::nat) − (x::nat))
using b by auto
    then show ∃ k. select l (Suc x) j = select k 0 (j − Suc x) by simp
  qed
qed

lemma selectlenhelp3: ∀ j. select l 0 j = 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** ∃ *k*. *select l i j* = *select k 0* (*j*−*i*) **by** *simp*
then have ∃ *k*. *length* (*select l i j*) = *length* (*select k 0* (*j*−*i*)) **by** *auto*
then have *c*: ∃ *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*: ⋀ *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* ⇒ *nat* **where**
zerocount: *zerocount n* = ((((*n* + *64*) *div 512*) + *1*) ∗ *512*) − *n* − (*65*::*nat*)

definition *helppadd* :: *bv* ⇒ *bv* ⇒ *nat* ⇒ *bv* **where**
helppadd x y n = *x* @ [*One*] @ *zerolist* (*zerocount n*) @ *zerolist* (*64* − *length y*) @*y*

definition *sha1padd* :: *bv* ⇒ *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 ⇒ bit list ⇒ bit list ⇒ bit list ⇒ 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 ⇒ 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: ⋀n xs :: bit list. length (delblockhelp xs n) <= length xs
  proof –
    fix n and xs :: bit list
    show length (delblockhelp xs n) ≤ length xs
      by (induct n rule: delblockhelp.induct) auto
  qed
  have ⋀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 (λ(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*: $(y \ mod \ (a::nat) = z \ mod \ a) \Longrightarrow (x*y) \ mod \ a = (x*z) \ mod \ a$
by (*rule mod-mult-cong* [*OF refl*])

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

end

6 Positive differences

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

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

lemma *timesdistributesoverpdifference*:
$m*(pdifference \ a \ b) = pdifference \ (m*(a::nat)) \ (m* \ (b::nat))$
by (*auto simp add*: *nat-distrib*)

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

lemma *invers*: $a \le x \Longrightarrow (x::nat) = x - a + a$
by *auto*

lemma *invers2*: $[\![a \le b; (b-a) = p*q]\!] \Longrightarrow (b::nat) = a+p*q$
apply (*subst addconst* [*symmetric*])
apply (*assumption*)
apply (*subst add.commute*, *rule invers*, *simp*)
done

lemma *modadd*: $[\![b = a+p*q]\!] \Longrightarrow (a::nat) \ mod \ p = b \ mod \ p$
by *auto*

lemma *equalmodstrick1*: $pdifference \ a \ b \ mod \ p = 0 \Longrightarrow a \ mod \ p = b \ mod \ p$
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*: $b \le c \Longrightarrow c - (c - b) = c - c + (b::nat)$
by *auto*

lemma *diff-add-assoc2*: $a \le c \Longrightarrow c - (c - a + b) = (c - c + (a::nat) - b)$
apply (*subst diff-diff-left* [*symmetric*])
apply (*subst diff-add-assoc*)

```
  apply auto
  done

lemma diff-add-diff: x ≤ b ⟹ (b::nat) − x + y − b = y − x
  by (induct b) auto

lemma equalmodstrick2:
  assumes a mod p = b mod p
  shows pdifference a b mod p = 0
proof −
  { fix a b
    assume *: a mod p = b mod p
    have a − b = a div p * p + a mod p − b div p * p − b mod p
      by simp
    also have ... = a div p * p − b div p * p
      using * by (simp only:)
    also have ... = (a div p − b div p) * p
      by (simp add: diff-mult-distrib)
    finally have (a − b) mod p = 0
      by simp
  }
  from this [OF assms] this [OF assms [symmetric]]
  show ?thesis by simp
qed

lemma primekeyrewrite:
  fixes p::nat shows ⟦prime p; p dvd (a*b);~(p dvd a)⟧ ⟹ p dvd b
  apply (subst (asm) prime-dvd-mult-nat)
  apply auto
  done

lemma multzero: ⟦0 < m mod p; m*a = 0⟧ ⟹ (a::nat) = 0
  by auto

lemma primekeytrick:
  fixes A B :: nat
  assumes (M * A) mod P = (M * B) mod P
  assumes M mod P ≠ 0 and prime P
  shows A mod P = B mod P
proof −
  from assms have M > 0
    by (auto intro: ccontr)
  from assms have *: ⋀q. P dvd M * q ⟹ P dvd q
     using primekeyrewrite [of P M] unfolding dvd-eq-mod-eq-0 [symmetric] by
blast
  from equalmodstrick2 [OF assms(1)] ‹M > 0› 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*: $[\![x\ mod\ a = (0::nat);\ x\ mod\ b = 0;\ prime\ a;\ prime\ b;\ a \neq b]\!] \Longrightarrow x\ mod\ (a*b) = 0$
  by (*simp add*: *mod-eq-0-iff-dvd primes-coprime divides-mult*)

lemma *specializedtoprimes1*:
  fixes *p*::*nat*
  shows $[\![prime\ p;\ prime\ q;\ p \neq q;\ a\ mod\ p = b\ mod\ p\ ;\ a\ mod\ q = b\ mod\ q]\!]$
    $\Longrightarrow a\ mod\ (p*q) = b\ mod\ (p*q)$
by (*metis equalmodstrick1 equalmodstrick2 productdivides*)

lemma *specializedtoprimes1a*:
  fixes *p*::*nat*
  shows $[\![prime\ p;\ prime\ q;\ p \neq q;\ a\ mod\ p = b\ mod\ p;\ a\ mod\ q = b\ mod\ q;\ b < p*q\ ]\!]$
  $\Longrightarrow a\ mod\ (p*q) = b$
  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* ⇒ *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*:

```
  nat-to-bv-length x y ≠ [] ⟹ length (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 ≠ [] ⟹ bv-to-nat (nat-to-bv-length x y) = x
  unfolding nat-to-bv-length by auto


definition
  roundup :: nat ⇒ nat ⇒ nat where
  roundup: roundup x y = (if (x mod y = 0) then (x div y) else (x div y) + 1)


lemma rnddvd: b dvd a ⟹ roundup a b * b = a
  by (auto simp add: roundup dvd-eq-mod-eq-0)

lemma bv-to-nat-zero-prepend: bv-to-nat a = bv-to-nat (0#a)
  by auto

primrec remzero:: bv ⇒ bv where
  remzero [] = []
| remzero (a#b) = (if a = 1 then (a#b) else remzero b)

lemma remzeroeq: bv-to-nat a = bv-to-nat (remzero a)
proof (induct a)
  show bv-to-nat [] = bv-to-nat (remzero [])
    by simp
next
  case (Cons a1 a2)
  show bv-to-nat a2 = bv-to-nat (remzero a2) ⟹
       bv-to-nat (a1 # a2) = bv-to-nat (remzero (a1 # a2))
  proof (cases a1)
    assume a: a1=0 then have bv-to-nat (a1#a2) = bv-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-to-bv a)
proof (auto)
  assume b: nat-to-bv a = []
  have a: bv-to-nat [] = 0 by simp
  have c: bv-to-nat (nat-to-bv a) = 0 using a and b by simp
  from x have d: bv-to-nat (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: remzero ((replicate n 0)@l) = remzero l
by (induct n, auto)

lemma length-bvxor-bound: a <= length l ⟹ a <= length (bvxor l l2)
proof (induct a)
  case 0
  then show ?case by simp
next
  case (Suc a)
  have a: Suc a ≤ length l by fact
  with Suc.hyps have b: a ≤ length (bvxor l l2) by simp
  show Suc a ≤ length (bvxor l l2)
  proof cases
    assume c: a = length (bvxor l l2)
    show Suc a ≤ length (bvxor l l2)
    proof (simp add: bvxor)
      have length l <= max (length l) (length l2) by simp
      then show Suc a ≤ max (length l) (length l2) using a by simp
    qed
  next
    assume a ≠ length (bvxor l l2)
    then have a < length (bvxor l l2) using b by simp
    then show ?thesis by simp
  qed
qed

lemma nat-to-bv-helper-legacy-induct:
  (⋀n. n ≠ (0::nat) ⟶ P (n div 2) ⟹ P n) ⟹ P x
unfolding atomize-imp[symmetric]
by (induction-schema, simp, lexicographic-order)

lemma len-lower-bound:
  assumes 0 < n
  shows 2^(length (nat-to-bv n) − Suc 0) ≤ 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: Suc 0 < (n::nat) and b: ~Suc 0<n div 2
    then have n=2 ∨ n=3
    proof (cases n<=3)
      assume n<=3 and Suc 0<n
      then show n=2 ∨ n=3 by auto
    next
      assume ~n<=3 then have 3<n by simp
```

then have *1 < n div 2* **by** *arith*
then show *n=2* ∨ *n=3* **using** *b* **by** *simp*
qed
then show *2* ^(*length* (*nat-to-bv-helper n* []) − *Suc 0*) ≤ *n*
proof (*cases n=2*)
assume *a*: *n=2* **then have** *nat-to-bv-helper n* [] = [**1**, **0**]
proof −
have *nat-to-bv-helper n* [] = *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* ^(*length* (*nat-to-bv-helper n* []) − *Suc 0*) ≤ *n* **using** *a* **by** *simp*
next
assume *n=2* ∨ *n=3* **and** *n~=2*
then have *a*: *n=3* **by** *simp*
then have *nat-to-bv-helper n* [] = [**1**, **1**]
proof −
have *nat-to-bv-helper n* [] = *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*^(*length* (*nat-to-bv-helper n* []) − *Suc 0*) <=*n* **using** *a* **by** *simp*
qed
next
fix *n*
assume *a*: *Suc 0<n* **and**
b: *2* ^(*length* (*nat-to-bv-helper* (*n div 2*) []) − *Suc 0*) ≤ *n div 2*
have (*2*::*nat*) ^(*length* (*nat-to-bv-helper n* []) − *Suc 0*) =
2^(*length* (*nat-to-bv-helper* (*n div 2*) []) + *1* − *Suc 0*)
proof −
have *length* (*nat-to-bv n*) = *length* (*nat-to-bv* (*n 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*::*nat*)^(*length* (*nat-to-bv-helper* (*n div 2*) []) + *1* − *Suc 0*)
=
2^(*length* (*nat-to-bv-helper* (*n div 2*) []) − *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** *nat-to-bv n ~= []*
proof −
from *nat-to-bv-non0[of n]* **have** *∃x. nat-to-bv n = x@[if n mod 2 = 0 then* **0** *else* **1**] **using** *a* **by** *simp*
then show *?thesis* **by** *auto*
qed

lemma *hd-append*: *x ~= [] ⟹ hd (x @ xs) = hd x*
by *(induct x) auto*

lemma *hd-one*: *0<n ⟹ hd (nat-to-bv-helper n []) =* **1**

```
proof (induct n rule: nat-to-bv-helper-legacy-induct)
  fix n
  assume *: n ≠ 0 ⟶ 0 < n div 2 ⟶ hd (nat-to-bv-helper (n div 2) []) = 1
    and 0 < n
  show hd (nat-to-bv-helper n []) = 1
  proof (cases 1<n)
    assume a: 1<n
    with * have b: 0 < n div 2 ⟶ hd (nat-to-bv-helper (n div 2) []) = 1 by simp
    from a have c: 0<n div 2 by arith
    then have d: hd (nat-to-bv-helper (n div 2) []) = 1 using b by simp
    also from a have 0<n by simp
    then have hd (nat-to-bv-helper n []) =
        hd (nat-to-bv (n div 2) @ [if n mod 2 = 0 then 0 else 1])
      using nat-to-bv-def and nat-to-bv-non0[of n] by auto
    then have hd (nat-to-bv-helper n []) =
        hd (nat-to-bv (n 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 hd (nat-to-bv-helper n []) = hd (nat-to-bv-helper (n div 2) [])
      using nat-to-bv-def by simp
    then show hd (nat-to-bv-helper n []) = 1 using b and c by simp
  next
    assume ~1 < n with ‹0<n› have c: n=1 by simp
    have nat-to-bv-helper 1 [] = [1] by (simp add: nat-to-bv-helper.simps)
    then show hd (nat-to-bv-helper n []) = 1 using c by simp
  qed
qed

lemma prime-hd-non-zero:
  fixes p::nat assumes a: prime p and b: prime q shows hd (nat-to-bv (p*q)) ~=
0
proof -
  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 primerew: fixes p::nat shows  ⟦m dvd p; m~=1; m~=p⟧ ⟹ ~ prime p
  by (auto simp add: prime-nat-iff)


lemma two-dvd-exp: 0<x ⟹ (2::nat) dvd 2^x
  by (induct x) auto

lemma exp-prod1: ⟦1<b;2^x=2*(b::nat)⟧ ⟹ 2 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 primerew[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 ⟹ xs ≠ [] ⟹ 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 ⇒ nat ⇒ 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* ⇒ *bv* ⇒ *bv*
where *generate-M′ mHash salt-new* = *bv-prepend 64* **0** [] @ *mHash* @ *salt-new*

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

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

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

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

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

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

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

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

definition *MGF*:: *bv* ⇒ *nat* ⇒ *bv*
where
MGF Z l = (*if l* = *0* ∨ *2^32*∗(*length* (*sha1 Z*)) < *l*
then []
else MGF1 Z (*roundup l* (*length* (*sha1 Z*)) − *1*) *l*)

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

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

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

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

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

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

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

definition *emsapss-encode-help2*:: *bv* ⇒ *nat* ⇒ *bv*
where *emsapss-encode-help2 M′ emBits* = *emsapss-encode-help3* (*sha1 M′*) *em-Bits*

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

definition *emsapss-decode-help11*:: *bv* ⇒ *bv* ⇒ *bool*
where *emsapss-decode-help11 H′ H* = (*if H′* ≠ *H then False else True*)

definition *emsapss-decode-help10*:: *bv* ⇒ *bv* ⇒ *bool*
where *emsapss-decode-help10 M′ H* = *emsapss-decode-help11* (*sha1 M′*) *H*

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

definition *emsapss-decode-help8*:: *bv* ⇒ *bv* ⇒ *bv* ⇒ *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 (*bvxor 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* ≤ *length M* ∨ *2^32*∗*160*<*emBits*)
then False
else emsapss-decode-help1 (*sha1 M*) *EM emBits*)

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

lemma *roundup-ge-emBits*: $0 < emBits \Longrightarrow 0 < x \Longrightarrow emBits \le (roundup\ emBits\ x) * x$
  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*: $0 < emBits \Longrightarrow 0 < x \Longrightarrow 0 \le roundup\ emBits\ x * x - emBits$
  by (*simp add*: *roundup*)

lemma *roundup-le-7*: $0 < emBits \Longrightarrow roundup\ emBits\ 8 * 8 - emBits \le\ 7$
  by (*auto simp add*: *roundup*) *arith*

lemma *roundup-nat-ge-8-help*:
  $length\ (sha1\ M) + sLen + 16 \le emBits \Longrightarrow 8 \le roundup\ emBits\ 8 * 8 - (length\ (sha1\ M) + 8)$
  apply (*insert roundup-ge-emBits* [*of emBits 8*])
  apply (*simp add*: *roundup sha1len sLen-def*)
  done

lemma *roundup-nat-ge-8*:
  $length\ (sha1\ M) + sLen + 16 \le emBits \Longrightarrow 8 \le roundup\ emBits\ 8 * 8 - (length\ (sha1\ M) + 8)$
  apply (*insert roundup-nat-ge-8-help* [*of M emBits*])
  apply *arith*
  done

lemma *roundup-le-ub*:
  $[\![\ 176 + sLen \le emBits;\ emBits \le 2\hat{\ }32 * 160]\!] \Longrightarrow (roundup\ emBits\ 8) * 8 - 168 \le 2\hat{\ }32 * 160$
  apply (*simp add*: *roundup*)
  apply (*safe*)
  apply (*simp*)
  apply (*arith*)+
  done

lemma *modify-roundup-ge1*: $[\![8 \le roundup\ emBits\ 8 * 8 - 168]\!] \Longrightarrow 176 \le roundup\ emBits\ 8 * 8$
  by *arith*

lemma *modify-roundup-ge2*: ⟦ *176* ≤ *roundup emBits 8* ∗ *8*⟧ ⟹ *21* < *roundup emBits 8*
  by *simp*

lemma *roundup-help1*: ⟦ *0* < *roundup l 160*⟧ ⟹ (*roundup l 160* − *1*) + *1* = (*roundup l 160*)
  by *arith*

lemma *roundup-help1-new*: ⟦ *0* < *l*⟧ ⟹ (*roundup l 160* − *1*) + *1* = (*roundup l 160*)
  apply (*drule roundup-positiv* [*of l*])
  apply *arith*
  done

lemma *roundup-help2*: ⟦*176* + *sLen* ≤ *emBits*⟧ ⟹ *roundup emBits 8* ∗ *8* − *emBits* <= *roundup emBits 8* ∗ *8* − *160* − *sLen* − *16*
  by (*simp add*: *sLen-def*)

lemma *bv-prepend-equal*: *bv-prepend* (*Suc n*) *b l* = *b*#*bv-prepend n b l*
  by (*simp add*: *bv-prepend*)

lemma *length-bv-prepend*: *length* (*bv-prepend n b l*) = *n*+*length l*
  by (*induct n*) (*simp-all add*: *bv-prepend*)

lemma *length-bv-prepend-drop*: *a* <= *length xs* ⟶ *length* (*bv-prepend a b* (*drop a xs*)) = *length xs*
  by (*simp add*:*length-bv-prepend*)

lemma *take-bv-prepend*: *take n* (*bv-prepend n b x*) = *bv-prepend n b* []
  by (*induct n*) (*simp add*: *bv-prepend*)+

lemma *take-bv-prepend2*: *take n* (*bv-prepend n b xs*@*ys*@*zs*) = *bv-prepend n b* []
  by (*induct n*) (*simp add*: *bv-prepend*)+

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

lemma *bv-prepend-append2*:
  x < *y* ⟹ *bv-prepend y b xs* = (*bv-prepend x b* [])@(*bv-prepend* (*y*−*x*) *b* [])@*xs*
  by (*simp add*: *bv-prepend replicate-add* [*symmetric*])

lemma *drop-bv-prepend-help2*: ⟦*x* < *y*⟧ ⟹ *drop x* (*bv-prepend y b* []) = *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*: ⟦*x* = *y*⟧ ⟹ *drop x* (*bv-prepend y b* []) = *bv-prepend* (*y*−*x*) *b* []

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

lemma *drop-bv-prepend-help4*: ⟦$x \leq y$⟧ ⟹ *drop x* (*bv-prepend y b* []) = *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*: *bv-prepend x b* [] @ *bv-prepend y b* [] = *bv-prepend* ($x + y$) *b* []
by (*induct x*) (*simp add*: *bv-prepend*)+

lemma *bv-prepend-drop*: $x \leq y$ ⟶ *bv-prepend x b* (*drop x* (*bv-prepend y b* [])) = *bv-prepend y 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*: *bv-prepend x b* (*left* @ *right*) = *bv-prepend x b left* @ *right*
by (*induct x*) (*simp add*: *bv-prepend*)+

lemma *length-generate-DB*: *length* (*generate-DB PS*) = *length PS* + *8* + *sLen*
by (*simp add*: *generate-DB-def sLen-def*)

lemma *length-generate-PS*: *length* (*generate-PS emBits 160*) = (*roundup emBits 8*)∗*8* − *sLen* − *160* − *16*
by (*simp add*: *generate-PS-def length-bv-prepend*)

lemma *length-bvxor*: *length a* = *length b* ⟹ *length* (*bvxor a b*) = *length a*
by (*simp add*: *bvxor*)

lemma *length-MGF2*: *length* (*MGF2 Z m*) = *Suc m* ∗ *length* (*sha1* (*Z* @ *nat-to-bv-length m 32*))
by (*induct m*) (*simp*+, *simp add*: *sha1len*)

lemma *length-MGF1*: *l* ≤ (*Suc n*) ∗ *160* ⟹ *length* (*MGF1 Z n l*) = *l*
by (*simp add*: *MGF1-def length-MGF2 sha1len*)

lemma *length-MGF*: *0* < *l* ⟹ *l* ≤ *2*^*32* ∗ *length* (*sha1 x*) ⟹ *length* (*MGF x 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*:
⟦ *0* < *emBits*; *length* (*sha1 M*) + *sLen* + *16* ≤ *emBits*⟧

⟹ *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*:
⟦ *roundup emBits 8* ∗ *8* − *emBits* ≤ *length maskedDB*⟧
⟹ *length* (*maskedDB-zero maskedDB emBits*) = *length maskedDB*
by (*simp add*: *maskedDB-zero-def length-bv-prepend*)

lemma *take-equal-bv-prepend*:
⟦ *176* + *sLen* ≤ *emBits*; *roundup emBits 8* ∗ *8* − *emBits* ≤ *7*⟧
⟹ *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*) **0** []
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 + *sLen* ≤ *emBits* ⟹ *roundup emBits 8* ∗ *8* − *emBits* ≤ *roundup emBits 8* ∗ *8* − (*176* + *sLen*)
⟹ *0* = *roundup emBits 8* ∗ *8* − *emBits* − (*roundup emBits 8* ∗ *8* − (*176* + *sLen*))
by *arith*

lemma *get-salt*: ⟦ *176* + *sLen* ≤ *emBits*; *roundup emBits 8* ∗ *8* − *emBits* ≤ *7*⟧ ⟹ (*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*: ⟦*roundup emBits 8* ∗ *8* − *emBits* ≤ *length x*; (*roundup emBits 8*) ∗ *8* − (*length* (*sha1 M*)) − *8* = *length* (*maskedDB-zero x emBits*)⟧ ⟹ *generate-maskedDB* (*maskedDB-zero x emBits* @ *y* @ *z*) *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*: ⟦ *roundup emBits 8 ∗ 8 − emBits ≤ length x*; *length* (*maskedDB-zero x emBits*) = (*roundup emBits 8*) ∗ *8 − 168*; *length y = 160*⟧ ⟹ *generate-H* (*maskedDB-zero x emBits @ y @ z*) *emBits 160 = y*
  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*: [|*roundup emBits 8 ∗ 8 − emBits* <= *roundup emBits 8 ∗ 8 −* (*176 + sLen*); *length* (*generate-PS emBits 160*) = *roundup emBits 8 ∗ 8 −* (*176 + sLen*)|] ==> *length* (*bv-prepend* (*roundup emBits 8 ∗ 8 − emBits*) **0** (*drop* (*roundup emBits 8 ∗ 8 − emBits*) (*generate-PS emBits 160*))) = *length* (*generate-PS emBits 160*)
  by (*simp add*: *length-bv-prepend-drop*)

lemma *x01-elim*: ⟦*176 + sLen ≤ emBits*; *roundup emBits 8 ∗ 8 − emBits ≤ 7*⟧ ⟹ *take 8* (*drop* (*roundup emBits 8 ∗ 8 −* (*length* (*sha1 M*) + *sLen* + *16*))(*maskedDB-zero* (*generate-DB* (*generate-PS emBits 160*)) *emBits*)) = [**0**, **0**, **0**, **0**, **0**, **0**, **0**, **1**]
 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* <= *length x length x = length y*
  shows *drop n* (*bv-mapzip f x y*) = *bv-mapzip f* (*drop n x*) (*drop n y*)
proof −
  have ⋀*x y*. *n* <= *length x* ⟹ *length x = length y* ⟹
      drop n (*bv-mapzip f x y*) = *bv-mapzip f* (*drop n x*) (*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 *length a = length b*
  shows *bvxor* (*bvxor a b*) *b = a*
proof −
  have ⋀*b*. *length a = length b* ⟹ *bvxor* (*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* <= *length a* **and** *length a* = *length b*
shows *bv-prepend x* **0** (*drop x* (*bvxor* (*bv-prepend x* **0** (*drop x* (*bvxor a b*))) *b*)) = *bv-prepend x* **0** (*drop x a*)
proof −
have *drop x* (*bvxor* (*bv-prepend x* **0** (*drop x* (*bvxor a b*))) *b*) = *drop x a*
apply (*unfold bvxor 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-bvxor* [*of drop x a drop x b*])
apply (*subgoal-tac length* (*replicate x* **0** @ *drop x* (*bv-mapzip* ($\oplus_b$) *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* ($\oplus_b$) *a b*)) *b* ($\oplus_b$)])
apply (*simp*)
apply (*insert drop-bv-mapzip* [*of x a b* ($\oplus_b$)])
apply (*simp*)
apply (*fold bvxor*)
apply (*simp-all*)
done
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *bvxorxor-elim*: ⟦ *roundup emBits 8* ∗ *8* − *emBits* ≤ *length a*; *length a* = *length b*⟧ ⟹ (*maskedDB-zero* (*bvxor* (*maskedDB-zero* (*bvxor a b*) *emBits*)*b*) *emBits*) = *bv-prepend* (*roundup emBits 8* ∗ *8* − *emBits*) **0** (*drop* (*roundup emBits 8* ∗ *8* − *emBits*) *a*)
by (*simp add*: *maskedDB-zero-def bvxorxor-elim-help*)

lemma *verify*: ⟦(*emsapss-encode M emBits*) ≠ []; *EM*=(*emsapss-encode M emBits*)⟧ ⟹ *emsapss-decode M EM emBits* = *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 emBits 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 bvxorxor-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*: *emsapss-encode m x* ≠ [] ⟹ *bv-to-nat* (*emsapss-encode m x*) < *2*^(*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*
(*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*
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*: ⋀ *a b c*. *length* (*a*@*b*@*c*) = *length a* + *length b* + *length c* **by** *simp*
from *a* **and** *b* **show** *?thesis* **using** *c* **by** *simp*
qed

lemma *MGFLen-helper*: *MGF z l* ~= [] ⟹ *l* <= *2*^*32*∗(*length* (*sha1 z*))
proof (*cases 2*^*32*∗*length* (*sha1 z*) < *l*)
assume *x*: *MGF z l* ~= []
assume *a*: *2* ^ *32* ∗ *length* (*sha1 z*) < *l*
then have *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
  (
  (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
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*)))))) ≤ *2^32* ∗ *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* ((*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*))))))))
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-bvxor-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*
(*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*))
< *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*
≤ (*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* ~= *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* <= ((*length* (*nat-to-bv* (*p*∗*q*)) − *Suc 0*) *div 8* + *1*)∗*8* **by** *auto*
ultimately show *length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*
≤ (*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* ≤ *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*)) <= *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*: *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* **using** *length-helper1* **by** *simp*
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** *p* **and** *q* **and** *length-helper2* **and** *mgf* **and** *len* **by** *simp*
from *a* **and** *b* **and** *c* **have** *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*) **by** *simp*
then have *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*) **using** *round* **by** *simp*
moreover have *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*) **using** *x* **and** *emBits-roundup-cancel2* **by** *simp*
moreover have *0*<*length* (*nat-to-bv* (*p*∗*q*))
proof −
from *p* **have** *s*: *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** *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*: ⟦*prime p*; *prime q*; *emsapss-encode m* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) ≠ []; (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *mod 8* ~= *0*⟧ ⟹ *length* (*remzero* (*emsapss-encode m* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*))) < *length* (*nat-to-bv* (*p*∗*q*))
proof −
assume *a*: *emsapss-encode m* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) ≠ []
assume *p*: *prime p* **and** *q*: *prime q*
assume *x*: (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *mod 8* ~= *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* ≤ *length m*) ∨ (*2^32* ∗ *160* < (*length* (*nat-to-bv* (*p*∗*q*)) − *Suc 0*)))
then []
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 []
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* <= (*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* []
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-def emsapss-encode-help1-def*)
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-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 emsapss-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*)))))) ~=
[] **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*)
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-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)))))))
(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
(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)) **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 ~= q **and** c: emsapss-encode m (length (nat-to-bv (p ∗ q)) − Suc 0) ≠ []
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* ^(*roundup* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *8* ∗ *8*) < *p*∗*q*
proof −
from *d* **have** *e*: *roundup* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *8* ∗ *8* = *length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0* **using** *rnddvd* **by** *simp*
have *p*∗*q* = *bv-to-nat* (*nat-to-bv* (*p*∗*q*)) **by** *simp*
then have *2* ^(*length* (*nat-to-bv* (*p* ∗ *q*)) − *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*^(*length* (*nat-to-bv* (*p*∗*q*)) − *Suc 0*) ~= *p*∗*q*
proof (*cases 2*^(*length* (*nat-to-bv* (*p*∗*q*)) − *Suc 0*) = *p*∗*q*)
assume *2*^(*length* (*nat-to-bv* (*p*∗*q*)) − *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*^(*length* (*nat-to-bv* (*p*∗*q*)) − *Suc 0*) ~= *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* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)) < *2* ^(*roundup* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)*8* ∗ *8*)
using *bv-to-nat-emsapss-encode-le* [*of m* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)] **by** *auto*
ultimately show *?thesis* **by** *simp*
next
assume *y*: ~(*8 dvd* (*length* (*nat-to-bv* (*p*∗*q*)) − *Suc 0*))
then show *?thesis*
proof −
from *y* **have** *x*: ~((*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *mod 8* = *0*) **by** (*simp add*: *dvd-eq-mod-eq-0*)
from *remzeroeq* **have** *d*: *bv-to-nat* (*emsapss-encode m* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)) = *bv-to-nat* (*remzero* (*emsapss-encode m* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*))) **by** *simp*
from *a* **and** *b* **and** *c* **and** *x* **and** *length-emsapss-smaller-pq*[*of p q m*] **have** *bv-to-nat* (*remzero* (*emsapss-encode m* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*))) < *bv-to-nat* (*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 *bv-to-nat* (*emsapss-encode m* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)) < *p* ∗ *q* **using** *d* **and** *bv-nat-bv* **by** *simp*

qed
qed
qed

lemma *rsa-pss-verify*: $[\![$ *prime p*; *prime q*; $p \neq q$; $n = p*q$; $e*d$ *mod* ((*pred p*)$*$(*pred q*)) = *1*; *rsapss-sign m e n* $\neq$ []; *s* = *rsapss-sign m e n* $]\!] \Longrightarrow$ *rsapss-verify m s d n* = *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.