

RSAPSS

Christina Lindenberg and Kai Wirt
Darmstadt Technical University
Cryptography and Computeralgebra

December 12, 2009

Abstract

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

Contents

1	Extensions to the Isabelle Word theory required for SHA1	2
2	Message Padding for SHA1	4
3	Formal definition of the secure hash algorithm	4
4	Definition of rsacrypt	7
5	Lemmata for modular arithmetic	7
6	Positive differences	8
7	Lemmata for modular arithmetic with primes	9
8	Pigeon hole principle	10
9	Fermats little theorem	14
10	Correctness proof for RSA	16
11	Extensions to the Word theory required for PSS	16
12	EMSA-PSS encoding and decoding operation	18

1 Extensions to the Isabelle Word theory required for SHA1

```
theory WordOperations
imports Word
begin
```

```
types bv = bit list
```

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

definition

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

definition

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

definition

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

primrec last where

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

primrec dellast where

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

fun 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)))
```

$(n-(1::nat))$
 $| \text{select } (x\#l) \ i \ n = (\text{if } (i \leq 0) \ \text{then } (\text{selecthelp } (x\#l) \ n) \ \text{else } \text{select } l \ (i-(1::nat)))$
 $(n-(1::nat))$

definition

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

definition

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

primrec zerolist where

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

primrec hextovb where

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

primrec hexvtobv where

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

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

lemma $\text{selectlenhelp2}: \bigwedge i. \text{ALL } l \ j. \text{EX } k. \text{select } l \ i \ j = \text{select } k \ 0 \ (j - i)$
 $\langle \text{proof} \rangle$

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

lemma $\text{selectlen}: \text{length } (\text{select } l \ i \ j) = j - i + 1$
 $\langle \text{proof} \rangle$

lemma *addmod32len*: $\bigwedge a b. \text{length} (\text{addmod32 } a b) = 32$
 <proof>

end

2 Message Padding for SHA1

theory *SHA1Padding*
imports *WordOperations*
begin

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

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

```

<proof>

termination *<proof>*

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>

```

```

lemma sha1len: length (sha1 m) = 160
<proof>

```

```

end

```

4 Definition of rsacrypt

```

theory Crypt
imports Mod
begin

```

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

```

definition
  even :: nat ⇒ bool where
    even n ⟷ 2 dvd n

```

```

fun rsa-crypt :: nat ⇒ nat ⇒ nat ⇒ nat
where
  rsa-crypt M 0 n = 1
| rsa-crypt M (Suc e) n = (if even (Suc e)
  then (rsa-crypt M (Suc e div 2) n) ^ 2 mod n
  else (M * ((rsa-crypt M (Suc e div 2) n) ^ 2 mod n)) mod n)

```

```

lemma div-2-times-2: (if (even m) then (m div 2 * 2 = m) else (m div 2 * 2 =
m - 1))
<proof>

```

```

theorem cryptcorrect: n ≠ 0 ⇒ n ≠ 1 ⇒ rsa-crypt M e n = M^e mod n
<proof>

```

```

end

```

5 Leammata for modular arithmetic

```

theory Mod
imports Main
begin

```

lemma *divmultassoc*: $a \text{ div } (b*c) * (b*c) = ((a \text{ div } (b * c)) * b)*(c::nat)$
 ⟨proof⟩

lemma *delmod*: $(a::nat) \text{ mod } (b*c) \text{ mod } c = a \text{ mod } c$
 ⟨proof⟩

lemma *timesmod1*: $((x::nat)*(y::nat) \text{ mod } n) \text{ mod } (n::nat) = ((x*y) \text{ mod } n)$
 ⟨proof⟩

lemma *timesmod3*: $((a \text{ mod } (n::nat)) * b) \text{ mod } n = (a*b) \text{ mod } n$
 ⟨proof⟩

lemma *remainderexplemma*: $(y \text{ mod } (a::nat) = z \text{ mod } a) \implies (x*y) \text{ mod } a = (x*z) \text{ mod } a$
 ⟨proof⟩

lemma *remainderexp*: $((a \text{ mod } (n::nat))^i) \text{ mod } n = (a^i) \text{ mod } n$
 ⟨proof⟩

end

6 Positive differences

theory *Pdifference*

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

begin

definition

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

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

lemma *addconst*: $a = (b::nat) \implies c+a = c+b$
 ⟨proof⟩

lemma *invers*: $a \leq x \implies (x::nat) = x - a + a$
 ⟨proof⟩

lemma *invers2*: $\llbracket a \leq b; (b-a) = p*q \rrbracket \implies (b::nat) = a+p*q$
 ⟨proof⟩

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

lemma *equalmodstrick1*: $pdifference a b \text{ mod } p = 0 \implies a \text{ mod } p = b \text{ mod } p$

<proof>

lemma *diff-add-assoc*: $b \leq c \implies c - (c - b) = c - c + (b::nat)$
<proof>

lemma *diff-add-assoc2*: $a \leq c \implies c - (c - a + b) = (c - c + (a::nat) - b)$
<proof>

lemma *diff-add-diff*: $x \leq b \implies (b::nat) - x + y - b = y - x$
<proof>

lemma *equalmodstrick2*: $a \bmod p = b \bmod p \implies p \text{ difference } a \bmod p = 0$
<proof>

lemma *primekeyrewrite*: $\llbracket \text{prime } p; p \text{ dvd } (a*b); \sim(p \text{ dvd } a) \rrbracket \implies p \text{ dvd } b$
<proof>

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

lemma *primekeytrick*: $\llbracket (M*A) \bmod P = (M*B) \bmod P; M \bmod P \neq 0; \text{prime } P \rrbracket$
 $\implies A \bmod P = (B::nat) \bmod P$
<proof>

end

7 Lemmata for modular arithmetic with primes

theory *Productdivides*

imports *Pdifference*

begin

lemma *productdivides-lemma*: $\llbracket x \bmod z = (0::nat) \rrbracket \implies ((y*x) \bmod (y*z) = 0)$
<proof>

lemma *productdivides*: $\llbracket x \bmod a = (0::nat); x \bmod b = 0; \text{prime } a; \text{prime } b; a \neq b \rrbracket \implies x \bmod (a*b) = 0$
<proof>

lemma *specializedtoprimes1*: $\llbracket \text{prime } p; \text{prime } q; p \neq q; a \bmod p = b \bmod p; a \bmod q = b \bmod q \rrbracket$
 $\implies a \bmod (p*q) = b \bmod (p*q)$
<proof>

lemma *specializedtoprimes1a*:
 $\llbracket \text{prime } p; \text{prime } q; p \neq q; a \bmod p = b \bmod p; a \bmod q = b \bmod q; b < p*q \rrbracket$
 $\implies a \bmod (p*q) = b$
<proof>

end

8 Pigeon hole principle

theory *Pigeonholeprinciple*
imports *Productdivides*
begin

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

primrec *alldistinct* :: *nat list* \Rightarrow *bool*

where

alldistinct [] = *True*
| *alldistinct* ($x \# xs$) = ($\neg x \text{ mem } xs \wedge \text{alldistinct } xs$)

primrec *allesseq* :: *nat list* \Rightarrow *nat* \Rightarrow *bool*

where

allesseq [] n = *True*
| *allesseq* ($x \# xs$) n = ($x \leq n \wedge \text{allesseq } xs \ n$)

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

where

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

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

where

positives 0 = []
| *positives* (*Suc* n) = *Suc* $n \# \text{positives } n$

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

where

timeslist [] = 1
| *timeslist* ($x \# xs$) = $x * \text{timeslist } xs$

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

where

fac 0 = 1
| *fac* (*Suc* n) = *Suc* $n * \text{fac } n$

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

where

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

lemma *length-del*: $x \text{ mem } xs \implies \text{length } (\text{del } x \text{ } xs) < \text{length } xs$
⟨proof⟩

function *pigeonholeinduction*

where

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

termination ⟨proof⟩

lemma *old-pig-induct*:

fixes P

assumes P []

and $(\bigwedge (x::\text{nat}) \text{ } xs::\text{nat list.}$

$\neg \text{length } (x \# xs) \text{ mem } xs \longrightarrow P \text{ } xs \implies$

$\text{length } (x \# xs) \text{ mem } xs \longrightarrow P \text{ } (\text{del } (\text{length } (x \# xs)) (x \# xs)) \implies$

$P \text{ } (x \# xs)$)

shows P x

⟨proof⟩

definition

perm :: nat list \Rightarrow nat list \Rightarrow bool **where**

perm xs ys \longleftrightarrow sort xs = sort ys

lemma *allnonzerodelete*: $\text{allnonzero } xs \implies \text{allnonzero } (\text{del } x \text{ } xs)$

⟨proof⟩

lemma *notmemnotdelmem*: $x \neq a \implies \neg a \text{ mem } xs \implies \neg a \text{ mem } (\text{del } x \text{ } xs)$

⟨proof⟩

lemma *alldistinctdelete*: $\text{alldistinct } xs \implies \text{alldistinct } (\text{del } x \text{ } xs)$

⟨proof⟩

lemma *pigeonholeprinciple-lemma2*: $\neg (\text{Suc } n) \text{ mem } xs \implies \text{alldistinct } xs \text{ } (\text{Suc } n)$

$\implies \text{alldistinct } xs \text{ } n$

⟨proof⟩

lemma *pigeonholeprinciple-lemma1*:

$\text{alldistinct } xs \implies \text{alldistinct } xs \text{ } (\text{Suc } n) \implies \text{alldistinct } (\text{del } (\text{Suc } n) \text{ } xs) \text{ } n$

⟨proof⟩

lemma *anotinsort*: $a \neq x \implies a \text{ mem } b = a \text{ mem } (\text{insort } x \text{ } b)$

⟨proof⟩

lemma *ainsort*: $a \text{ mem } (\text{insort } a \text{ } b)$

<proof>

lemma *memesort*: $x \text{ mem } xs = x \text{ mem } (\text{sort } xs)$

<proof>

lemma *permmember*: $\llbracket \text{perm } xs \text{ } ys; x \text{ mem } xs \rrbracket \implies x \text{ mem } ys$

<proof>

lemma *alldistinctdelete*: $\llbracket \text{alldistinct } (x\#xs); \text{alldistinct } (x\#xs) (\text{length}(x\#xs)) \rrbracket$

$\implies \text{alldistinct } (\text{del } (\text{length}(x\#xs)) (x\#xs)) (\text{length } xs)$

<proof>

lemma *perminsert*: $\text{perm } xs \text{ } ys \implies \text{perm } (a\#xs) (a\#ys)$

<proof>

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

<proof>

lemma *dellengthinalldistinct*:

$\llbracket \text{alldistinct } (x\#xs); \text{alldistinct } (x\#xs) (\text{length } (x\#xs)); \text{length } (x\#xs) \text{ mem } xs \rrbracket$

$\implies \text{alldistinct } (\text{del } (\text{length } (x\#xs)) (x\#xs)) (\text{length } (\text{del } (\text{length } (x\#xs)) (x\#xs)))$

<proof>

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

<proof>

lemma *permSucLengthdel*:

$\llbracket \text{Suc } (\text{length } xs) \text{ mem } xs; \text{perm } (\text{positives } (\text{length } xs)) (x \# \text{del } (\text{Suc } (\text{length } xs)) xs);$

$x \neq \text{Suc } (\text{length } xs) \rrbracket \implies$

$\text{perm } (\text{Suc } (\text{length } xs) \# \text{positives } (\text{length } xs)) ((\text{Suc } (\text{length } xs)) \# (x \# \text{del } (\text{Suc } (\text{length } xs)) xs))$

<proof>

lemma *insortsym*: $\text{insort } a (\text{insort } x xs) = \text{insort } x (\text{insort } a xs)$

<proof>

lemma *insortsortdel*: $x \text{ mem } xs \implies \text{insort } x (\text{sort } (\text{del } x xs)) = (\text{sort } xs)$

<proof>

lemma *permSucLengthdel2*:

$\llbracket \text{Suc } (\text{length } xs) \text{ mem } xs; x \neq \text{Suc } (\text{length } xs);$

$\text{perm } (\text{Suc } (\text{length } xs) \# \text{positives } (\text{length } xs)) ((\text{Suc } (\text{length } xs)) \# (x \# \text{del } (\text{Suc } (\text{length } xs)) xs)) \rrbracket$

$\implies \text{perm } (\text{Suc } (\text{length } xs) \# \text{positives } (\text{length } xs)) (x \# xs)$

<proof>

lemma *dellengthinperm*:

$\llbracket \text{length } (x \# xs) \text{ mem } (x\#xs);$

$\text{perm } (\text{positives } (\text{length } (\text{del } (\text{length } (x \# xs)) (x \# xs)))) (\text{del } (\text{length } (x \# xs)) (x \# xs))$
 $\implies \text{perm } (\text{positives } (\text{length } (x \# xs))) (x \# xs)$
 <proof>

lemma positiveseq: $\text{positives } (\text{length } xs) = \text{rev } ([1 \ ..< \text{Suc}(\text{length } xs)])$
 <proof>

lemma memsetpositives:
 $\llbracket \text{perm } (\text{positives } (\text{length } xs)) \ x; \ 0 < x; \ x \leq \text{length } xs \rrbracket \implies x \in \text{set } (\text{positives } (\text{length } xs))$
 <proof>

lemma pigeonholeprinciple:
 $\text{allnonzero } xs \implies \text{alldistinct } xs \implies \text{allessseq } xs \ (\text{length } xs) \implies \text{perm } (\text{positives } (\text{length } xs)) \ xs$
 <proof>

thm notE [of $\text{allnonzero } (\text{del } (\text{length } (x \# xs)) (x \# xs))$]
 <proof>

lemmas seteqmem = mem-iff [symmetric]

lemma pigeonholeprinciple:
 $\text{allnonzero } xs \implies \text{alldistinct } xs \implies \text{allessseq } xs \ (\text{length } xs) \implies \text{perm } (\text{positives } (\text{length } xs)) \ xs$
 <proof>

lemma equaltimeslist: $\llbracket \text{sort } xs = \text{sort } ys \rrbracket \implies \text{timeslist } (\text{sort } xs) = \text{timeslist } (\text{sort } ys)$
 <proof>

lemma timeslistmultkom: $\text{timeslist } (xs) * x = x * \text{timeslist } (xs)$
 <proof>

lemma timeslistinsort: $\text{timeslist } (\text{insort } a \ xs) = \text{timeslist } (a \# xs)$
 <proof>

lemma timeslistseq: $\text{timeslist } (\text{sort } xs) = \text{timeslist } xs$
 <proof>

lemma permtimeslist: $\text{perm } xs \ ys \implies \text{timeslist } xs = \text{timeslist } ys$
 <proof>

lemma timeslistpositives: $\text{timeslist } (\text{positives } n) = \text{fac } n$
 <proof>

lemma pdvdnot: $\llbracket \text{prime } p; \neg p \ \text{dvd } x; \neg p \ \text{dvd } y \rrbracket \implies \neg p \ \text{dvd } x * y$
 <proof>

lemma *lessdvdnot*: $\llbracket \text{Suc } (x::\text{nat}) < p \rrbracket \implies \neg p \text{ dvd } \text{Suc } x$
<proof>

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

lemma *primefact*: $\text{prime } p \implies (n::\text{nat}) < p \implies \text{fac } n \text{ mod } p \neq 0$
<proof>

end

9 Fermats little theorem

theory *Fermat*
imports *Pigeonholeprinciple*
begin

primrec *pred*:: $\text{nat} \Rightarrow \text{nat}$
where
 pred 0 = 0
 | *pred* (Suc a) = a

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

lemma *remaindertimeslist*: $\text{timeslist } (S \ n \ M \ p) \text{ mod } p = \text{fac } n * M ^ n \text{ mod } p$
<proof>

lemma *sucassoc*: $(P + P * w) = P * \text{Suc } w$
<proof>

lemma *modI*: $0 < (x::\text{nat}) \text{ mod } p \implies 0 < x$
<proof>

lemma *delmulmod*: $\llbracket 0 < x \text{ mod } p; a < (b::\text{nat}) \rrbracket \implies x * a < x * b$
<proof>

lemma *swaple*: $(c < b) \implies ((a::\text{nat}) \leq b - c) \implies c \leq b - a$
<proof>

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

lemma *sucleI*: $\text{Suc } x \leq 0 \implies \text{False}$
<proof>

lemma *diffI*: $(0::nat) = b - b$
⟨proof⟩

lemma *alldistincts*: $prime\ p \implies (m\ mod\ p \neq 0) \implies (n2 < n1) \implies (n1 < p)$
 \implies
 $\neg(((m*n1)\ mod\ p)\ mem\ (S\ n2\ m\ p))$
⟨proof⟩

lemma *alldistincts2*: $prime\ p \implies m\ mod\ p \neq 0 \implies n < p \implies alldistinct\ (S\ n\ m\ p)$
⟨proof⟩

lemma *notdvdless*: $\neg\ a\ dvd\ b \implies 0 < (b::nat)\ mod\ a$
⟨proof⟩

lemma *allnonzerop*: $prime\ p \implies m\ mod\ p \neq 0 \implies n < p \implies allnonzero\ (S\ n\ m\ p)$
⟨proof⟩

lemma *predI*: $a < p \implies a \leq pred\ p$
⟨proof⟩

lemma *predd*: $pred\ p = p - (1::nat)$
⟨proof⟩

lemma *alldesseqps*: $p \neq 0 \implies alldesseq\ (S\ n\ m\ p)\ (pred\ p)$
⟨proof⟩

lemma *lengths*: $length\ (S\ n\ m\ p) = n$
⟨proof⟩

lemma *suconeless*: $prime\ p \implies p - 1 < p$
⟨proof⟩

lemma *primenotzero*: $prime\ p \implies p \neq 0$
⟨proof⟩

lemma *onemodprime*: $prime\ p \implies 1\ mod\ p = (1::nat)$
⟨proof⟩

lemma *fermat*: $\llbracket prime\ p; m\ mod\ p \neq 0 \rrbracket \implies m^{(p-(1::nat))}\ mod\ p = 1$
⟨proof⟩

end

10 Correctness proof for RSA

```
theory Cryptinverts
imports Fermat Crypt
begin
```

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

```
lemma cryptinverts-hilf1: prime p  $\implies (m * m^{(k * pred p)}) \bmod p = m \bmod p$ 
  <proof>
```

```
lemma cryptinverts-hilf2: prime p  $\implies m * (m^{(k * (pred p) * (pred q))}) \bmod p =$ 
  m mod p
  <proof>
```

```
lemma cryptinverts-hilf3: prime q  $\implies m * (m^{(k * (pred p) * (pred q))}) \bmod q =$ 
  m mod q
  <proof>
```

```
lemma cryptinverts-hilf4:
  [[prime p; prime q; p  $\neq$  q; m < p*q; x mod ((pred p)*(pred q)) = 1]]  $\implies m^x$ 
  mod (p*q) = m
  <proof>
```

```
lemma primmultgreater: [[prime p; prime q; p  $\neq$  2; q  $\neq$  2]]  $\implies 2 < p*q$ 
  <proof>
```

```
lemma primmultgreater2: [[prime p; prime q; p  $\neq$  q]]  $\implies 2 < p*q$ 
  <proof>
```

```
lemma cryptinverts: [[prime p; prime q; p  $\neq$  q; n = p*q; m < n;
  e*d mod ((pred p)*(pred q)) = 1]]  $\implies \text{rsa-crypt (rsa-crypt m e n) d n} = m$ 
  <proof>
```

```
end
```

11 Extensions to the Word theory required for PSS

```
theory Wordarith
imports WordOperations ~~/src/HOL/Old-Number-Theory/Primes
begin
```

```
definition
  nat-to-bv-length :: nat  $\Rightarrow$  nat  $\Rightarrow$  bv where
```

```
  nat-to-bv-length:
  nat-to-bv-length n l = (if length(nat-to-bv n)  $\leq$  l then bv-extend l 0 (nat-to-bv n)
```

else [])

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

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

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

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

definition

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

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

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

<proof>

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

<proof>

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

remzero [] = []

| *remzero* (a#b) = (if a = 1 then (a#b) else *remzero* b)

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

<proof>

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

<proof>

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

<proof>

lemma *length-bvxor-bound*: $a \leq \text{length } l \implies a \leq \text{length } (\text{bxor } l \ l)$

<proof>

lemma *len-lower-bound*:

assumes $0 < n$

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

<proof>

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

<proof>

lemma *nat-to-bv-non-empty*: **assumes** $a: 0 < n$ **shows** $\text{nat-to-bv } n \sim = []$

<proof>

lemma *hd-append*: $x \sim [] \implies \text{hd } (x @ xs) = \text{hd } x$
<proof>

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

lemma *prime-hd-non-zero*: **assumes** *a*: prime *p* **and** *b*: prime *q* **shows** $\text{hd } (\text{nat-to-bv } (p*q)) \sim \mathbf{0}$
<proof>

lemma *primerew*: $\llbracket m \text{ dvd } p; m \sim 1; m \sim p \rrbracket \implies \sim \text{prime } p$
<proof>

lemma *two-dvd-exp*: $0 < x \implies (2::\text{nat}) \text{ dvd } 2^x$
<proof>

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

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

lemma *odd-mul-odd*: $\llbracket (2::\text{nat}) \text{ dvd } p; \sim 2 \text{ dvd } q \rrbracket \implies \sim 2 \text{ dvd } p*q$
<proof>

lemma *prime-equal*: $\llbracket \text{prime } p; \text{prime } q; 2^x = p*q \rrbracket \implies (p=q)$
<proof>

lemma *nat-to-bv-length-bv-to-nat*:
 $\text{length } xs = n \implies xs \neq [] \implies \text{nat-to-bv-length } (\text{bv-to-nat } xs) n = xs$
<proof>

end

12 EMSA-PSS encoding and decoding operation

theory *EMSA_PSS*
imports *SHA1 Wordarith*
begin

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

consts *BC*:: *bv*
salt:: *bv*
sLen:: *nat*

```

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

```

defs

```

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

```

```

BC:
BC == [One, Zero, One, One, One, One, Zero, Zero]

```

```

salt:
salt == []

```

```

sLen:
sLen == length salt

```

```

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

```

```

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

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

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

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

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

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

MGF:
MGF Z l == if l = 0 ∨ 232*(length (sha1 Z)) < l
then []
else MGF1 Z (roundup l (length (sha1 Z)) - 1) l

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

emsapss-encode:
emsapss-encode M emBits == if (264 ≤ length M ∨ 232 * 160 < emBits)
then []
else emsapss-encode-help1 (sha1 M) emBits

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

emsapss-encode-help2:
emsapss-encode-help2 M' emBits == emsapss-encode-help3 (sha1 M') emBits

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

```

```

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

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

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

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

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

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

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

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

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

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

```

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

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

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

then *False*
else *emsapss-decode-help8* *mHash* *DB-zero* *H*

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

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

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

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

primrec

MGF2 *Z* 0 = *sha1* (*Z*@(*nat-to-bv-length* 0 32))
MGF2 *Z* (*Suc* *n*) = (*MGF2* *Z* *n*)@(*sha1* (*Z*@(*nat-to-bv-length* (*Suc* *n*) 32)))

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

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

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

lemma *roundup-le-7*: 0 < *emBits* ⇒ *roundup* *emBits* 8 * 8 - *emBits* ≤ 7

<proof>

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

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

<proof>

lemma *roundup-nat-ge-8*:

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

<proof>

lemma *roundup-le-ub*:

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

lemma *length-bv-prepend*: $\text{length } (\text{bv-prepend } n \ b \ l) = n + \text{length } l$

<proof>

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

<proof>

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

<proof>

lemma *take-bv-prepend2*: $\text{take } n \text{ (bv-prepend } n \text{ b xs@ys@zs)} = \text{bv-prepend } n \text{ b []}$
 ⟨proof⟩

lemma *bv-prepend-append*: $\text{bv-prepend } a \text{ b } x = \text{bv-prepend } a \text{ b [] @ } x$
 ⟨proof⟩

lemma *bv-prepend-append2*:
 $x < y \implies \text{bv-prepend } y \text{ b } xs = (\text{bv-prepend } x \text{ b []}) @ (\text{bv-prepend } (y-x) \text{ b []}) @ xs$
 ⟨proof⟩

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

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

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

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

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

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

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

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

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

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

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

<proof>

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

<proof>

lemma *solve-length-generate-DB*:

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

$\implies \text{length}(\text{generate-DB}(\text{generate-PS } \text{emBits}(\text{length}(\text{sha1 } x)))) = (\text{roundup } \text{emBits } 8) * 8 - 168$

<proof>

lemma *length-maskedDB-zero*:

$\llbracket \text{roundup } \text{emBits } 8 * 8 - \text{emBits} \leq \text{length } \text{maskedDB} \rrbracket$

$\implies \text{length}(\text{maskedDB-zero } \text{maskedDB } \text{emBits}) = \text{length } \text{maskedDB}$

<proof>

lemma *take-equal-bv-prepend*:

$\llbracket 176 + \text{sLen} \leq \text{emBits}; \text{roundup } \text{emBits } 8 * 8 - \text{emBits} \leq 7 \rrbracket$

$\implies \text{take}(\text{roundup } \text{emBits } 8 * 8 - \text{length}(\text{sha1 } M) - \text{sLen} - 16)(\text{maskedDB-zero}(\text{generate-DB}(\text{generate-PS } \text{emBits } 160))) \text{emBits} =$

$\text{bv-prepend}(\text{roundup } \text{emBits } 8 * 8 - \text{length}(\text{sha1 } M) - \text{sLen} - 16) \mathbf{0} \llbracket$

<proof>

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

<proof>

lemma *equal-zero*:

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

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

<proof>

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

<proof>

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

$(\text{length}(\text{sha1 } M)) = \text{maskedDB-zero } x \text{ } \text{emBits}$

<proof>

lemma *generate-H-elim*: $\llbracket \text{roundup } \text{emBits } 8 * 8 - \text{emBits} \leq \text{length } x; \text{length}(\text{maskedDB-zero } x \text{ } \text{emBits}) = (\text{roundup } \text{emBits } 8) * 8 - 168; \text{length } y = 160 \rrbracket$

$\implies \text{generate-H}(\text{maskedDB-zero } x \text{ } \text{emBits} @ y @ z) \text{emBits } 160 = y$

<proof>

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

lemma *x01-elim*: $\llbracket 176 + sLen \leq emBits; \text{roundup } emBits \ 8 * 8 - emBits \leq 7 \rrbracket \implies \text{take } 8 \ (\text{drop } (\text{roundup } emBits \ 8 * 8 - (\text{length } (sha1 \ M) + sLen + 16)) (\text{maskedDB}\text{-}zero \ (generate\text{-}DB \ (generate\text{-}PS \ emBits \ 160)) \ emBits)) = [\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{1}]$
 ⟨proof⟩

lemma *drop-bv-mapzip*:
assumes $n \leq \text{length } x \ \text{length } x = \text{length } y$
shows $\text{drop } n \ (bv\text{-}mapzip \ f \ x \ y) = bv\text{-}mapzip \ f \ (\text{drop } n \ x) \ (\text{drop } n \ y)$
 ⟨proof⟩

lemma [*simp*]:
assumes $\text{length } a = \text{length } b$
shows $bvxor \ (bvxor \ a \ b) \ b = a$
 ⟨proof⟩

lemma *bvxorxor-elim-help*:
assumes $x \leq \text{length } a$ **and** $\text{length } a = \text{length } b$
shows $bv\text{-}prepend \ x \ \mathbf{0} \ (\text{drop } x \ (bvxor \ (bv\text{-}prepend \ x \ \mathbf{0} \ (\text{drop } x \ (bvxor \ a \ b))) \ b))$
 =
 $bv\text{-}prepend \ x \ \mathbf{0} \ (\text{drop } x \ a)$
 ⟨proof⟩

lemma *bvxorxor-elim*: $\llbracket \text{roundup } emBits \ 8 * 8 - emBits \leq \text{length } a; \text{length } a = \text{length } b \rrbracket \implies (\text{maskedDB}\text{-}zero \ (bvxor \ (\text{maskedDB}\text{-}zero \ (bvxor \ a \ b) \ emBits) \ b) \ emBits) = bv\text{-}prepend \ (\text{roundup } emBits \ 8 * 8 - emBits) \ \mathbf{0} \ (\text{drop } (\text{roundup } emBits \ 8 * 8 - emBits) \ a)$
 ⟨proof⟩

lemma *verify*: $\llbracket (\text{emsapss}\text{-}encode \ M \ emBits) \neq []; EM = (\text{emsapss}\text{-}encode \ M \ emBits) \rrbracket \implies \text{emsapss}\text{-}decode \ M \ EM \ emBits = True$
 ⟨proof⟩

end

13 RSS-PSS encoding and decoding operation

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

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

```

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

```

defs

```

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

```

```

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

```

```

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

```

lemma length-emsapss-encode:

```

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

```

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

<proof>

lemma length-helper1: **shows** length

```

(bvxor
 (generate-DB
 (generate-PS (length (nat-to-bv (p * q)) - Suc 0)
 (length (sha1 (generate-M' (sha1 m) salt))))))
 (MGF (sha1 (generate-M' (sha1 m) salt))
 (length
 (generate-DB
 (generate-PS (length (nat-to-bv (p * q)) - Suc 0)
 (length (sha1 (generate-M' (sha1 m) salt)))))) @
 sha1 (generate-M' (sha1 m) salt) @ BC)
 = length
 (bxor
 (generate-DB
 (generate-PS (length (nat-to-bv (p * q)) - Suc 0)
 (length (sha1 (generate-M' (sha1 m) salt))))))

```

$(MGF (sha1 (generate-M' (sha1 m) salt)))$
 $(length$
 $(generate-DB$
 $(generate-PS (length (nat-to-bv (p * q)) - Suc 0)$
 $(length (sha1 (generate-M' (sha1 m) salt)))))) + 168$
 $\langle proof \rangle$

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

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

and mgf : $(MGF (sha1 (generate-M' (sha1 m) salt)))$

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

lemma *emBits-roundup-cancel*: $emBits \bmod 8 \sim = 0 \implies (roundup emBits 8) * 8 - emBits = 8 - (emBits \bmod 8)$
 $\langle proof \rangle$

lemma *emBits-roundup-cancel2*: $emBits \bmod 8 \sim = 0 \implies (roundup emBits 8) * 8 - (8 - (emBits \bmod 8)) = emBits$
 $\langle proof \rangle$

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

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

(length
 (generate-DB
 (generate-PS (length (nat-to-bv (p * q)) - Suc 0)
 (length (sha1 (generate-M' (sha1 m) salt)))))))))
 ⟨proof⟩

lemma *length-helper*: **assumes** *p*: prime *p* **and** *q*: prime *q* **and** *x*: (length (nat-to-bv (p * q)) - Suc 0) mod 8 $\sim =$ 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)))))) $\sim =$ []
and *len*: length (sha1 M) + sLen + 16 \leq (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⟩

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

lemma *bv-to-nat-emsapss-smaller-pq*: **assumes** *a*: prime *p* **and** *b*: prime *q* **and** *pneq*: $p \sim = q$ **and** *c*: emsapss-encode *m* (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⟩

lemma *rsa-pss-verify*: \llbracket prime *p*; prime *q*; $p \neq q$; $n = p * q$; $e * d \bmod ((\text{pred } p) * (\text{pred } q)) = 1$; rsapss-sign *m* *e* *n* \neq []; $s = \text{rsapss-sign } m \ e \ n \rrbracket \implies \text{rsapss-verify } m \ s \ d$
n = True
 ⟨proof⟩

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.