

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

1 Extensions to the Word theory required for SHA1

```
theory WordOperations
imports Word
begin
```

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

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

```
definition
```

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

```
definition
```

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

```
definition
```

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

```
primrec last where
```

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

```
primrec dellast where
```

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

```
fun bvrol where
```

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

```
fun bvrer where
```

```
  bvrer a 0 = a
| bvrer [] x = []
| bvrer x (Suc n) = bvrer (last x # dellast x) n
```

```
fun selecthelp where
```

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

```
fun select where
```

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

```
definition
```

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

definition

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

primrec zerolist where

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

primrec hextobv where

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

primrec hexvtobv where

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

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

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

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

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

lemma addmod32len: $\bigwedge a \ b. \text{length } (\text{addmod32 } a \ b) = 32$
 $\langle \text{proof} \rangle$

end

2 Message Padding for SHA1

```
theory SHA1Padding
imports WordOperations
begin
```

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

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

```
definition sha1padd :: bv  $\Rightarrow$  bv where
  sha1padd: sha1padd x = helppadd x (nat-to-bv (length x)) (length x)
```

```
end
```

3 Formal definition of the secure hash algorithm

```
theory SHA1
imports SHA1Padding
begin
```

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

```
definition fif where
  fif: fif x y z = bvor (bvand x y) (bvand (bv-not x) z)
```

```
definition fxor where
  fxor: fxor x y z = bxor (bxor 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 K_4 **where**

$K_4: K_4 = \text{hexvtobv } [xC, xA, x6, x2, xC, x1, xD, x6]$

definition $kselect :: \text{nat} \Rightarrow \text{bit list}$ **where**

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

definition $getblock$ **where**

$getblock: getblock\ x = \text{select } x\ 0\ 511$

fun $delblockhelp$ **where**

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

definition $delblock$ **where**

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

primrec $sha1compress$ **where**

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

definition $sha1expandhelp$ **where**

$sha1expandhelp\ x\ i = (\text{let } j = (79+16-i) \text{ in } (\text{bvrol } (\text{bvxor}(\text{bvxor}(\text{select } x\ (32*(j-(3::\text{nat})))\ (31+(32*(j-(3::\text{nat}))))\ (\text{select } x\ (32*(j-(8::\text{nat})))\ (31+(32*(j-(8::\text{nat}))))\ (\text{bvxor}(\text{select } x\ (32*(j-(14::\text{nat})))\ (31+(32*(j-(14::\text{nat}))))\ (\text{select } x\ (32*(j-(16::\text{nat})))\ (31+(32*(j-(16::\text{nat}))))\ 1))$

fun $sha1expand$ **where**

$sha1expand\ x\ i = (\text{if } (i < 16) \text{ then } x \text{ else}$
 $\text{let } y = sha1expandhelp\ x\ i \text{ in}$

$sha1expand (x @ y) (i - 1)$

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 $b\ []\ A\ B\ C\ D\ E = (let\ H = sha1compressstart\ 79\ b\ A\ B\ C\ D\ E;$

$AA = addmod32\ A\ (select\ H\ 0\ 31);$

$BB = addmod32\ B\ (select\ H\ 32\ 63);$

$CC = addmod32\ C\ (select\ H\ 64\ 95);$

$DD = addmod32\ D\ (select\ H\ 96\ 127);$

$EE = addmod32\ E\ (select\ H\ 128\ 159)$

$in\ AA@BB@CC@DD@EE)$

$| sha1block\ b\ x\ A\ B\ C\ D\ E = (let\ H = sha1compressstart\ 79\ b\ A\ B\ C\ D\ E;$

$AA = addmod32\ A\ (select\ H\ 0\ 31);$

$BB = addmod32\ B\ (select\ H\ 32\ 63);$

$CC = addmod32\ C\ (select\ H\ 64\ 95);$

$DD = addmod32\ D\ (select\ H\ 96\ 127);$

$EE = addmod32\ E\ (select\ H\ 128\ 159)$

$in\ sha1block\ (getblock\ x)\ (delblock\ x)\ AA\ BB\ CC\ DD\ E)$

$\langle proof \rangle$

termination $\langle proof \rangle$

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*: $length\ (sha1block\ b\ x\ A\ B\ C\ D\ E) = 160$

$\langle proof \rangle$

lemma *sha1len*: $length\ (sha1\ m) = 160$

<proof>

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* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *nat*
where

cryptcorrect: *rsa-crypt* *M e n* = $M \wedge e \text{ mod } n$

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* \wedge 2 mod *n*
else (*M* * *rsa-crypt* *M (e div 2) n* \wedge 2 mod *n*) mod *n*)

<proof>

end

5 Lemmata for modular arithmetic

theory *Mod*
imports *Main*
begin

lemma *divmultassoc*: $a \text{ div } (b*c) * (b*c) = ((a \text{ div } (b * c)) * b)*(c::nat)$
<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*: $((a \text{ mod } (n::nat)) \wedge i) \text{ mod } n = (a \wedge i) \text{ mod } n$
<proof>

end

6 Positive differences

theory *Pdifference*
imports *HOL-Computational-Algebra.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*: $a = (b::nat) \Longrightarrow c+a = c+b$

(*proof*)

lemma *invers*: $a \leq x \Longrightarrow (x::nat) = x - a + a$

(*proof*)

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

(*proof*)

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

(*proof*)

lemma *equalmodstrick1*: $pdifference\ a\ b \bmod p = 0 \Longrightarrow a \bmod p = b \bmod p$

(*proof*)

lemma *diff-add-assoc*: $b \leq c \Longrightarrow c - (c - b) = c - c + (b::nat)$

(*proof*)

lemma *diff-add-assoc2*: $a \leq c \Longrightarrow c - (c - a + b) = (c - c + (a::nat)) - b$

(*proof*)

lemma *diff-add-diff*: $x \leq b \Longrightarrow (b::nat) - x + y - b = y - x$

(*proof*)

lemma *equalmodstrick2*:

assumes $a \bmod p = b \bmod p$

shows $pdifference\ a\ b \bmod p = 0$

(*proof*)

lemma *primekeyrewrite*:

fixes $p::nat$ **shows** $\llbracket prime\ p; p\ dvd\ (a*b); \sim(p\ dvd\ a) \rrbracket \Longrightarrow p\ dvd\ b$

(*proof*)

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

lemma *primekeytrick*:
fixes $A B :: nat$
assumes $(M * A) \text{ mod } P = (M * B) \text{ mod } P$
assumes $M \text{ mod } P \neq 0$ **and** *prime* P
shows $A \text{ mod } P = B \text{ mod } P$
<proof>

end

7 Lemmata for modular arithmetic with primes

theory *Productdivides*
imports *Pdifference*
begin

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

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

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

end

8 Correctness proof for RSA

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

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

primrec *pred*: $nat \Rightarrow nat$
where
pred 0 = 0

| $\text{pred } (\text{Suc } a) = a$

lemma *pred-unfold*:

$\text{pred } n = n - 1$
 $\langle \text{proof} \rangle$

lemma *fermat*:

assumes $\text{prime } p \ m \ \text{mod } p \neq 0$
shows $m^{\wedge}(p-(1::\text{nat})) \ \text{mod } p = 1$
 $\langle \text{proof} \rangle$

lemma *cryptinverts-hilf1*: $\text{prime } p \implies (m * m^{\wedge}(k * \text{pred } p)) \ \text{mod } p = m \ \text{mod } p$
 $\langle \text{proof} \rangle$

lemma *cryptinverts-hilf2*: $\text{prime } p \implies m * (m^{\wedge}(k * (\text{pred } p) * (\text{pred } q))) \ \text{mod } p = m \ \text{mod } p$
 $\langle \text{proof} \rangle$

lemma *cryptinverts-hilf3*: $\text{prime } q \implies m * (m^{\wedge}(k * (\text{pred } p) * (\text{pred } q))) \ \text{mod } q = m \ \text{mod } q$
 $\langle \text{proof} \rangle$

lemma *cryptinverts-hilf4*:

$m^{\wedge} x \ \text{mod } (p * q) = m$ **if** $\text{prime } p \ \text{prime } q \ p \neq q$
 $m < p * q \ x \ \text{mod } (\text{pred } p * \text{pred } q) = 1$
 $\langle \text{proof} \rangle$

lemma *primmultgreater*: **fixes** $p::\text{nat}$ **shows** $\llbracket \text{prime } p; \text{prime } q; p \neq 2; q \neq 2 \rrbracket \implies 2 < p * q$
 $\langle \text{proof} \rangle$

lemma *primmultgreater2*: **fixes** $p::\text{nat}$ **shows** $\llbracket \text{prime } p; \text{prime } q; p \neq q \rrbracket \implies 2 < p * q$
 $\langle \text{proof} \rangle$

lemma *cryptinverts*: $\llbracket \text{prime } p; \text{prime } q; p \neq q; n = p * q; m < n; e * d \ \text{mod } ((\text{pred } p) * (\text{pred } q)) = 1 \rrbracket \implies \text{rsa-crypt } (\text{rsa-crypt } m \ e \ n) \ d \ n = m$
 $\langle \text{proof} \rangle$

end

9 Extensions to the Word theory required for PSS

theory *Wordarith*

imports *WordOperations HOL-Computational-Algebra.Primes*

begin

definition

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

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

lemma *length-nat-to-bv-length:*
nat-to-bv-length $x\ y \neq \square \implies \text{length} (\text{nat-to-bv-length } x\ y) = y$
 ⟨proof⟩

lemma *bv-to-nat-nat-to-bv-length:*
nat-to-bv-length $x\ y \neq \square \implies \text{bv-to-nat} (\text{nat-to-bv-length } x\ y) = x$
 ⟨proof⟩

definition
roundup $:: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
roundup: *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 $\square = \square$
 | *remzero* $(a\#b) = (\text{if } a = \mathbf{1} \text{ then } (a\#b) \text{ else } \text{remzero } b)$

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

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*: $a \leq \text{length } l \implies a \leq \text{length} (\text{bxor } l\ l2)$
 ⟨proof⟩

lemma *nat-to-bv-helper-legacy-induct*:
 $(\bigwedge n. n \neq (0::\text{nat}) \longrightarrow P (n \text{ div } 2) \implies P n) \implies P x$
 ⟨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*:
fixes $p::\text{nat}$ **assumes** a : *prime* p **and** b : *prime* q **shows** $\text{hd } (\text{nat-to-bv } (p*q)) \sim =$
 $\mathbf{0}$
<proof>

lemma *primerew*: **fixes** $p::\text{nat}$ **shows** $[[m \text{ dvd } p; m \sim = 1; m \sim = p]] \implies \sim \text{ prime } p$
<proof>

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

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

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

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

lemma *prime-equal*: **fixes** $p::\text{nat}$ **shows** $[[\text{prime } p; \text{prime } q; 2^{\wedge}x = p*q]] \implies (p=q)$
<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

10 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

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

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

definition *salt*:: bv
where *salt* = []

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

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

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

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

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

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

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

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

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

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

definition *MGF*:: $bv \Rightarrow nat \Rightarrow bv$

where

$MGF\ Z\ l = (if\ l = 0 \vee 2^{32} * (length\ (sha1\ Z)) < l$
 then \square
 else $MGF1\ Z\ (roundup\ l\ (length\ (sha1\ Z)) - 1)\ l)$

definition *emsapss-encode-help8*:: $bv \Rightarrow bv \Rightarrow bv$

where *emsapss-encode-help8* $DBzero\ H = DBzero\ @\ H\ @\ BC$

definition *emsapss-encode-help7*:: $bv \Rightarrow bv \Rightarrow nat \Rightarrow bv$

where *emsapss-encode-help7* $maskedDB\ H\ emBits =$
emsapss-encode-help8 $(maskedDB-zero\ maskedDB\ emBits)\ H$

definition *emsapss-encode-help6*:: $bv \Rightarrow bv \Rightarrow bv \Rightarrow nat \Rightarrow bv$

where *emsapss-encode-help6* $DB\ dbMask\ H\ emBits =$
 (if $dbMask = \square$
 then \square
 else *emsapss-encode-help7* $(bvxor\ DB\ dbMask)\ H\ emBits)$

definition *emsapss-encode-help5*:: $bv \Rightarrow bv \Rightarrow nat \Rightarrow bv$

where *emsapss-encode-help5* $DB\ H\ emBits =$
emsapss-encode-help6 $DB\ (MGF\ H\ (length\ DB))\ H\ emBits$

definition *emsapss-encode-help4*:: $bv \Rightarrow bv \Rightarrow nat \Rightarrow bv$

where *emsapss-encode-help4* $PS\ H\ emBits =$
emsapss-encode-help5 $(generate-DB\ PS)\ H\ emBits$

definition *emsapss-encode-help3*:: $bv \Rightarrow nat \Rightarrow bv$

where *emsapss-encode-help3* $H\ emBits =$
emsapss-encode-help4 $(generate-PS\ emBits\ (length\ H))\ H\ emBits$

definition *emsapss-encode-help2*:: $bv \Rightarrow nat \Rightarrow bv$

where *emsapss-encode-help2* $M'\ emBits = emsapss-encode-help3\ (sha1\ M')\ em-$
Bits

definition *emsapss-encode-help1*:: $bv \Rightarrow nat \Rightarrow bv$

where *emsapss-encode-help1* $mHash\ emBits =$
 (if $emBits < length\ (mHash) + sLen + 16$
 then \square
 else *emsapss-encode-help2* $(generate-M'\ mHash\ salt)\ emBits)$

definition *emsapss-encode*:: $bv \Rightarrow nat \Rightarrow bv$

where *emsapss-encode* $M\ emBits =$
 (if $(2^{64} \leq length\ M \vee 2^{32} * 160 < emBits)$
 then \square
 else *emsapss-encode-help1* $(sha1\ M)\ emBits)$

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

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

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

definition *emsapss-decode-help8*:: $bv \Rightarrow bv \Rightarrow bv \Rightarrow bool$
where *emsapss-decode-help8* $mHash\ DB-zero\ H =$
emsapss-decode-help9 $mHash\ (generate-salt\ DB-zero)\ H$

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

definition *emsapss-decode-help6*:: $bv \Rightarrow bv \Rightarrow bv \Rightarrow nat \Rightarrow bool$
where *emsapss-decode-help6* $mHash\ DB\ H\ emBits =$
emsapss-decode-help7 $mHash\ (maskedDB-zero\ DB\ emBits)\ H\ emBits$

definition *emsapss-decode-help5*:: $bv \Rightarrow bv \Rightarrow bv \Rightarrow bv \Rightarrow nat \Rightarrow bool$
where *emsapss-decode-help5* $mHash\ maskedDB\ dbMask\ H\ emBits =$
emsapss-decode-help6 $mHash\ (bvxor\ maskedDB\ dbMask)\ H\ emBits$

definition *emsapss-decode-help4*:: $bv \Rightarrow bv \Rightarrow bv \Rightarrow nat \Rightarrow bool$
where *emsapss-decode-help4* $mHash\ maskedDB\ H\ emBits =$
 $(if\ take\ ((roundup\ emBits\ 8)*8 - emBits)\ maskedDB \neq\ bv-prepend\ ((roundup$
 $emBits\ 8)*8 - emBits)\ \mathbf{0}\ []$
 $then\ False$
 $else\ emsapss-decode-help5\ mHash\ maskedDB\ (MGF\ H\ ((roundup\ emBits\ 8)*8$
 $- (length\ mHash) - 8))\ H\ emBits)$

definition *emsapss-decode-help3*:: $bv \Rightarrow bv \Rightarrow nat \Rightarrow bool$
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*:: $bv \Rightarrow bv \Rightarrow nat \Rightarrow bool$
where *emsapss-decode-help2* $mHash\ EM\ emBits =$
 $(if\ show-rightmost-bits\ EM\ 8 \neq\ BC$
 $then\ False$

else emsapss-decode-help3 mHash EM emBits)

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

lemma *roundup-positiv*: $0 < \text{emBits} \implies 0 < \text{roundup emBits } 160$
 <proof>

lemma *roundup-ge-emBits*: $0 < \text{emBits} \implies 0 < x \implies \text{emBits} \leq (\text{roundup emBits } x) * x$
 <proof>

lemma *roundup-ge-0*: $0 < \text{emBits} \implies 0 < x \implies 0 \leq \text{roundup emBits } x * x - \text{emBits}$
 <proof>

lemma *roundup-le-7*: $0 < \text{emBits} \implies \text{roundup emBits } 8 * 8 - \text{emBits} \leq 7$
 <proof>

lemma *roundup-nat-ge-8-help*:
 $\text{length (sha1 M) + sLen + 16} \leq \text{emBits} \implies 8 \leq \text{roundup emBits } 8 * 8 - (\text{length (sha1 M) + 8})$
 <proof>

lemma *roundup-nat-ge-8*:
 $\text{length (sha1 M) + sLen + 16} \leq \text{emBits} \implies 8 \leq \text{roundup emBits } 8 * 8 - (\text{length (sha1 M) + 8})$
 <proof>

lemma *roundup-le-ub*:
 $\llbracket 176 + \text{sLen} \leq \text{emBits}; \text{emBits} \leq 2^{32} * 160 \rrbracket \implies (\text{roundup emBits } 8) * 8 - 168 \leq 2^{32} * 160$
 <proof>

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

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

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

lemma *bv-prepend-equal*: $\text{bv-prepend } (Suc \ n) \ b \ l = b\#\text{bv-prepend } n \ b \ l$
<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*: $\text{take } n \ (\text{bv-prepend } n \ b \ xs@ys@zs) = \text{bv-prepend } n \ b \ []$
<proof>

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

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

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

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

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

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

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

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

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

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

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

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

lemma *length-MGF1*: $l \leq (Suc\ n) * 160 \implies length\ (MGF1\ Z\ n\ l) = l$
 ⟨proof⟩

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

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

lemma *length-maskedDB-zero*:
 [$roundup\ emBits\ 8 * 8 - emBits \leq length\ maskedDB$]
 $\implies length\ (maskedDB-zero\ maskedDB\ emBits) = length\ maskedDB$
 ⟨proof⟩

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

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

<proof>

lemma *equal-zero*:

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

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

lemma *drop-bv-mapzip*:

assumes $n \leq \text{length } x \ \text{length } x = \text{length } y$

shows $\text{drop } n (\text{bv-mapzip } f \ x \ y) = \text{bv-mapzip } f (\text{drop } n \ x) (\text{drop } n \ y)$

<proof>

lemma [*simp*]:

assumes $\text{length } a = \text{length } b$

shows $\text{bvxor } (\text{bvxor } a \ b) \ b = a$

<proof>

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

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

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

end

11 RSS-PSS encoding and decoding operation

theory *RSAPSS*
imports *EMSAPSS Cryptinverts*
begin

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

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

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

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

lemma *length-emsapss-encode*:
 $\text{emsapss-encode } m \text{ x} \neq \llbracket \implies \text{length } (\text{emsapss-encode } m \text{ x}) = \text{roundup } x 8 * 8$
<proof>

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

<proof>

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

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

<proof>

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*: $\text{length } (\text{sha1 } M) + sLen + 16 \leq (\text{length } (\text{nat-to-bv } (p * q))) - Suc \ 0$
shows *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*))))))
) = (*roundup* (*length* (*nat-to-bv* (*p * q*)) - *Suc 0*) 8) * 8 - 168

<proof>

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

<proof>

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

<proof>

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

<proof>

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

lemma *length-helper*: **assumes** *p*: prime *p* **and** *q*: prime *q* **and** *x*: $(length\ (nat-to-bv\ (p * q)) - Suc\ 0) \bmod 8 \sim = 0$ **and** *mgf*: $(MGF\ (sha1\ (generate-M'\ (sha1\ m)\ salt))\ salt) \sim = \llbracket$

$(length\ (generate-DB\ (generate-PS\ (length\ (nat-to-bv\ (p * q)) - Suc\ 0)\ (length\ (sha1\ (generate-M'\ (sha1\ m)\ salt)))))) \sim = \llbracket$
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*: $\llbracket \text{prime } p; \text{ prime } q; \text{ emsapss-encode } m \text{ (length (nat-to-bv } (p * q)) - \text{Suc } 0) \neq []; \text{ (length (nat-to-bv } (p * q)) - \text{Suc } 0) \bmod 8 \sim = 0 \rrbracket \implies \text{length (remzero (emsapss-encode } m \text{ (length (nat-to-bv } (p * q)) - \text{Suc } 0)) < \text{length (nat-to-bv } (p * q))$
<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*: $\llbracket \text{prime } p; \text{ prime } q; p \neq q; n = p * q; e * d \bmod ((\text{pred } p) * (\text{pred } q)) = 1; \text{ rsapss-sign } m \ e \ n \neq []; s = \text{rsapss-sign } m \ e \ n \rrbracket \implies \text{rsapss-verify } m \ s \ d$
n = 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.