# RSAPSS

Christina Lindenberg and Kai Wirt
Darmstadt Technical University
Cryptography and Computeralgebra

May 26, 2024

**Abstract**

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

## Contents

# 1 Extensions to the Word theory required for SHA1

**theory** *WordOperations*
**imports** *Word*
**begin**

**type-synonym** *bv* = *bit list*

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

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

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

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

**primrec** *last* **where**
  *last* [] = *Zero*
| *last* (*x*#*r*) = (*if* (*r*=[]) *then x else* (*last r*))

**primrec** *dellast* **where**
  *dellast* [] = []
| *dellast* (*x*#*r*) = (*if* (*r* = []) *then* [] *else* (*x*#*dellast r*))

**fun** *bvrol* **where**
  *bvrol a 0* = *a*
| *bvrol* [] *x* = []
| *bvrol* (*x*#*r*) (*Suc n*) = *bvrol* (*r*@[*x*]) *n*

**fun** *bvror* **where**
  *bvror a 0* = *a*
| *bvror* [] *x* = []
| *bvror x* (*Suc n*) = *bvror* (*last x* # *dellast x*) *n*

**fun** *selecthelp* **where**
  *selecthelp* [] *n* = (*if* (*n* <= *0*) *then* [*Zero*] *else* (*Zero* # *selecthelp* [] (*n*−(*1*::*nat*))))
| *selecthelp* (*x*#*l*) *n* = (*if* (*n* <= *0*) *then* [*x*] *else* (*x* # *selecthelp l* (*n*−(*1*::*nat*))))

**fun** *select* **where**
  *select* [] *i n* = (*if* (*i* <= *0*) *then* (*selecthelp* [] *n*) *else select* [] (*i*−(*1*::*nat*)) (*n*−(*1*::*nat*)))
| *select* (*x*#*l*) *i n* = (*if* (*i* <= *0*) *then* (*selecthelp* (*x*#*l*) *n*) *else select l* (*i*−(*1*::*nat*)) (*n*−(*1*::*nat*)))

**definition**

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

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

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

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

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

**lemma** *selectlenhelp*: *length* (*selecthelp l i*) = (*i* + *1*)
⟨*proof*⟩

**lemma** *selectlenhelp2*: $\bigwedge i.\ \forall l\ j.\ \exists k.$ *select l i j* = *select k 0* (*j* − *i*)
⟨*proof*⟩

**lemma** *selectlenhelp3*: $\forall j.$ *select l 0 j* = *selecthelp l j*
⟨*proof*⟩

**lemma** *selectlen*: *length* (*select l i j*) = *j* − *i* + *1*
⟨*proof*⟩

**lemma** *addmod32len*: $\bigwedge$ *a b*. *length* (*addmod32 a b*) = *32*
⟨*proof*⟩

**end**

## 2 Message Padding for SHA1

**theory** *SHA1Padding*
**imports** *WordOperations*
**begin**

**definition** *zerocount* :: *nat* ⇒ *nat* **where**
*zerocount*: *zerocount n* = ((((*n* + *64*) *div 512*) + *1*) ∗ *512*) − *n* − (*65*::*nat*)

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

**definition** *sha1padd* :: *bv* ⇒ *bv* **where**
*sha1padd*: *sha1padd x* = *helppadd x* (*nat-to-bv* (*length x*)) (*length x*)

**end**

## 3 Formal definition of the secure hash algorithm

**theory** *SHA1*
**imports** *SHA1Padding*
**begin**

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

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

**definition** *fxor* **where**
*fxor*: *fxor x y z* = *bvxor* (*bvxor x y*) *z*

**definition** *fmaj* **where**
*fmaj*: *fmaj x y z* = *bvor* (*bvor* (*bvand x y*) (*bvand x z*)) (*bvand y z*)

**definition** *fselect* :: *nat* ⇒ *bit list* ⇒ *bit list* ⇒ *bit list* ⇒ *bit list* **where**
*fselect*: *fselect r x y z* = (*if* (*r* < *20*) *then* (*fif x y z*) *else*
(*if* (*r* < *40*) *then* (*fxor x y z*) *else*
(*if* (*r* < *60*) *then* (*fmaj x y z*) *else*
(*fxor x y z*))))

**definition** *K1* **where**
*K1*: *K1* = *hexvtobv* [*x5*,*xA*,*x8*,*x2*,*x7*,*x9*,*x9*,*x9*]

**definition** *K2* **where**
*K2*: *K2* = *hexvtobv* [*x6*,*xE*,*xD*,*x9*,*xE*,*xB*,*xA*,*x1*]

**definition** *K3* **where**
*K3*: *K3* = *hexvtobv* [*x8*,*xF*,*x1*,*xB*,*xB*,*xC*,*xD*,*xC*]

**definition** *K4* **where**
  *K4*: *K4* = *hexvtobv* [*xC*,*xA*,*x6*,*x2*,*xC*,*x1*,*xD*,*x6*]

**definition** *kselect* :: *nat* ⇒ *bit list* **where**
  *kselect*: *kselect r* = (*if* (*r* < *20*) *then K1 else*
      (*if* (*r* < *40*) *then K2 else*
        (*if* (*r* < *60*) *then K3 else*
          *K4*)))

**definition** *getblock* **where**
  *getblock*: *getblock x* = *select x 0 511*

**fun** *delblockhelp* **where**
  *delblockhelp* [] *n* = []
  | *delblockhelp* (*x#r*) *n* = (*if n* <= *0 then x#r else delblockhelp r* (*n*−(*1*::*nat*)))

**definition** *delblock* **where**
  *delblock*: *delblock x* = *delblockhelp x 512*

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

**definition** *sha1expandhelp* **where**
  *sha1expandhelp x i* = (*let j* = (*79*+*16*−*i*) *in* (*bvrol* (*bvxor*(*bvxor*(
    *select x* (*32*∗(*j*−(*3*::*nat*))) (*31*+(*32*∗(*j*−(*3*::*nat*))))) (*select x* (*32*∗(*j*−(*8*::*nat*)))
(*31*+(*32*∗(*j*−(*8*::*nat*)))))) (*bvxor*(*select x* (*32*∗(*j*−(*14*::*nat*))) (*31*+(*32*∗(*j*−(*14*::*nat*)))))
(*select x* (*32*∗(*j*−(*16*::*nat*))) (*31*+(*32*∗(*j*−(*16*::*nat*))))))) *1*))

**fun** *sha1expand* **where**
  *sha1expand x i* = (*if* (*i* < *16*) *then x else*
    *let y* = *sha1expandhelp x i in*

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

**definition** *sha1compressstart* **where**
*sha1compressstart*: *sha1compressstart r b A B C D E* = *sha1compress r* (*sha1expand b 79*) *A B C D E*

**function** (*sequential*) *sha1block* **where**
*sha1block b* [] *A B C D E* = (*let H* = *sha1compressstart 79 b A B C D E*;
*AA* = *addmod32 A* (*select H 0 31*);
*BB* = *addmod32 B* (*select H 32 63*);
*CC* = *addmod32 C* (*select H 64 95*);
*DD* = *addmod32 D* (*select H 96 127*);
*EE* = *addmod32 E* (*select H 128 159*)
*in AA@BB@CC@DD@EE*)
| *sha1block b x A B C D E* = (*let H* = *sha1compressstart 79 b A B C D E*;
*AA* = *addmod32 A* (*select H 0 31*);
*BB* = *addmod32 B* (*select H 32 63*);
*CC* = *addmod32 C* (*select H 64 95*);
*DD* = *addmod32 D* (*select H 96 127*);
*EE* = *addmod32 E* (*select H 128 159*)
*in sha1block* (*getblock x*) (*delblock x*) *AA BB CC DD E*)
⟨*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** *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** $\textit{rsa-crypt} :: \textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{nat}$
**where**
$\textit{cryptcorrect}$: $\textit{rsa-crypt}\ M\ e\ n = M\ \hat{}\ e\ \textit{mod}\ n$

**lemma** *rsa-crypt-code* [*code*]:
$\textit{rsa-crypt}\ M\ e\ n = (\textit{if}\ e = 0\ \textit{then}\ 1\ \textit{mod}\ n$
$\textit{else if even}\ e\ \textit{then rsa-crypt}\ M\ (e\ \textit{div}\ 2)\ n\ \hat{}\ 2\ \textit{mod}\ n$
$\textit{else}\ (M * \textit{rsa-crypt}\ M\ (e\ \textit{div}\ 2)\ n\ \hat{}\ 2\ \textit{mod}\ n)\ \textit{mod}\ n)$
⟨*proof*⟩

**end**

# 5 Leammata for modular arithmetic

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

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

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

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

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

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

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

**end**

## 6 Positive differences

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

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

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

**lemma** *addconst*: *a* = (*b*::*nat*) ⟹ *c*+*a* = *c*+*b*
  ⟨*proof*⟩

**lemma** *invers*: *a* ≤ *x* ⟹ (*x*::*nat*) = *x* − *a* + *a*
  ⟨*proof*⟩

**lemma** *invers2*: ⟦*a* ≤ *b*; (*b*−*a*) = *p*∗*q*⟧ ⟹ (*b*::*nat*) = *a*+*p*∗*q*
  ⟨*proof*⟩

**lemma** *modadd*: ⟦*b* = *a*+*p*∗*q*⟧ ⟹ (*a*::*nat*) *mod p* = *b mod p*
  ⟨*proof*⟩

**lemma** *equalmodstrick1*: *pdifference a b mod p* = *0* ⟹ *a mod p* = *b mod p*
  ⟨*proof*⟩

**lemma** *diff-add-assoc*: *b* ≤ *c* ⟹ *c* − (*c* − *b*) = *c* − *c* + (*b*::*nat*)
  ⟨*proof*⟩

**lemma** *diff-add-assoc2*: *a* ≤ *c* ⟹ *c* − (*c* − *a* + *b*) = (*c* − *c* + (*a*::*nat*) − *b*)
  ⟨*proof*⟩

**lemma** *diff-add-diff*: *x* ≤ *b* ⟹ (*b*::*nat*) − *x* + *y* − *b* = *y* − *x*
  ⟨*proof*⟩

**lemma** *equalmodstrick2*:
  **assumes** *a mod p* = *b mod p*
  **shows** *pdifference a b mod p* = *0*
⟨*proof*⟩

**lemma** *primekeyrewrite*:
  **fixes** *p*::*nat* **shows** ⟦*prime p*; *p dvd* (*a*∗*b*);~(*p dvd a*)⟧ ⟹ *p dvd b*
  ⟨*proof*⟩

**lemma** *multzero*: $[\![0 < m \ mod \ p;\ m{*}a = 0]\!] \Longrightarrow (a{::}nat) = 0$
⟨*proof*⟩

**lemma** *primekeytrick*:
**fixes** $A\ B :: nat$
**assumes** $(M * A)\ mod\ P = (M * B)\ mod\ P$
**assumes** $M\ mod\ P \neq 0$ **and** *prime P*
**shows** $A\ mod\ P = B\ mod\ P$
⟨*proof*⟩

**end**

# 7 Lemmata for modular arithmetic with primes

**theory** *Productdivides*
**imports** *Pdifference*
**begin**

**lemma** *productdivides*: $[\![x\ mod\ a = (0{::}nat);\ x\ mod\ b = 0;\ prime\ a;\ prime\ b;\ a \neq b]\!] \Longrightarrow x\ mod\ (a{*}b) = 0$
⟨*proof*⟩

**lemma** *specializedtoprimes1*:
**fixes** $p{::}nat$
**shows** $[\![prime\ p;\ prime\ q;\ p \neq q;\ a\ mod\ p = b\ mod\ p\ ;\ a\ mod\ q = b\ mod\ q]\!]$
$\Longrightarrow a\ mod\ (p{*}q) = b\ mod\ (p{*}q)$
⟨*proof*⟩

**lemma** *specializedtoprimes1a*:
**fixes** $p{::}nat$
**shows** $[\![prime\ p;\ prime\ q;\ p \neq q;\ a\ mod\ p = b\ mod\ p;\ a\ mod\ q = b\ mod\ q;\ b < p{*}q\ ]\!]$
$\Longrightarrow a\ mod\ (p{*}q) = b$
⟨*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**
$pred\ 0 = 0$

| *pred* (*Suc a*) = *a*

**lemma** *pred-unfold*:
  *pred n* = *n* − *1*
  ⟨*proof*⟩

**lemma** *fermat*:
  **assumes** *prime p m mod p* ≠ *0*
  **shows** *m*^(*p*−(*1*::*nat*)) *mod p* = *1*
⟨*proof*⟩

**lemma** *cryptinverts-hilf1*: *prime p* ⟹ (*m* ∗ *m* ^(*k* ∗ *pred p*)) *mod p* = *m mod p*
  ⟨*proof*⟩

**lemma** *cryptinverts-hilf2*: *prime p* ⟹ *m*∗(*m*^(*k* ∗ (*pred p*) ∗ (*pred q*))) *mod p* = *m mod p*
  ⟨*proof*⟩

**lemma** *cryptinverts-hilf3*: *prime q* ⟹ *m*∗(*m*^(*k* ∗ (*pred p*) ∗ (*pred q*))) *mod q* = *m mod q*
  ⟨*proof*⟩

**lemma** *cryptinverts-hilf4*:
  *m* ^ *x mod* (*p* ∗ *q*) = *m* **if** *prime p prime q p* ≠ *q*
    *m* < *p* ∗ *q x mod* (*pred p* ∗ *pred q*) = *1*
⟨*proof*⟩

**lemma** *primmultgreater*: **fixes** *p*::*nat* **shows** ⟦ *prime p*; *prime q*; *p* ≠ *2*; *q* ≠ *2*⟧ ⟹ *2* < *p*∗*q*
  ⟨*proof*⟩

**lemma** *primmultgreater2*: **fixes** *p*::*nat* **shows** ⟦*prime p*; *prime q*; *p* ≠ *q*⟧ ⟹ *2* < *p*∗*q*
  ⟨*proof*⟩

**lemma** *cryptinverts*: ⟦ *prime p*; *prime q*; *p* ≠ *q*; *n* = *p*∗*q*; *m* < *n*;
    *e*∗*d mod* ((*pred p*)∗(*pred q*)) = *1*⟧ ⟹ *rsa-crypt* (*rsa-crypt m e n*) *d n* = *m*
  ⟨*proof*⟩

**end**

# 9 Extensions to the Word theory required for PSS

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

**definition**
  *nat-to-bv-length* :: *nat* ⇒ *nat* ⇒ *bv* **where**

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

**lemma** *length-nat-to-bv-length*:
$\textit{nat-to-bv-length}\ x\ y \neq [] \Longrightarrow \textit{length}\ (\textit{nat-to-bv-length}\ x\ y) = y$
⟨*proof*⟩

**lemma** *bv-to-nat-nat-to-bv-length*:
$\textit{nat-to-bv-length}\ x\ y \neq [] \Longrightarrow \textit{bv-to-nat}\ (\textit{nat-to-bv-length}\ x\ y) = x$
⟨*proof*⟩

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

**lemma** *rnddvd*: $b\ \textit{dvd}\ a \Longrightarrow \textit{roundup}\ a\ b * b = a$
⟨*proof*⟩

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

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

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

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

**lemma** *remzero-replicate*: $\textit{remzero}\ ((\textit{replicate}\ n\ \mathbf{0})@l) = \textit{remzero}\ l$
⟨*proof*⟩

**lemma** *length-bvxor-bound*: $a <= \textit{length}\ l \Longrightarrow a <= \textit{length}\ (\textit{bvxor}\ l\ l2)$
⟨*proof*⟩

**lemma** *nat-to-bv-helper-legacy-induct*:
$(\bigwedge n.\ n \neq (0::\textit{nat}) \longrightarrow P\ (n\ \textit{div}\ 2) \Longrightarrow P\ n) \Longrightarrow P\ x$
⟨*proof*⟩

**lemma** *len-lower-bound*:
**assumes** $0 < n$
**shows** $2\hat{}(\textit{length}\ (\textit{nat-to-bv}\ n) - \textit{Suc}\ 0) \leq n$

⟨*proof*⟩

**lemma** *length-lower*: **assumes** *a*: *length a* < *length b* **and** *b*: (*hd b*) ~= **0** **shows** *bv-to-nat a* < *bv-to-nat b*
⟨*proof*⟩

**lemma** *nat-to-bv-non-empty*: **assumes** *a*: *0*<*n* **shows** *nat-to-bv n* ~= []
⟨*proof*⟩

**lemma** *hd-append*: *x* ~= [] ⟹ *hd* (*x* @ *xs*) = *hd x*
⟨*proof*⟩

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

**lemma** *prime-hd-non-zero*:
**fixes** *p*::*nat* **assumes** *a*: *prime p* **and** *b*: *prime q* **shows** *hd* (*nat-to-bv* (*p*∗*q*)) ~= **0**
⟨*proof*⟩

**lemma** *primerew*: **fixes** *p*::*nat* **shows** ⟦*m dvd p*; *m*~=*1*; *m*~=*p*⟧ ⟹ ~ *prime p*
⟨*proof*⟩

**lemma** *two-dvd-exp*: *0*<*x* ⟹ (*2*::*nat*) *dvd 2*^*x*
⟨*proof*⟩

**lemma** *exp-prod1*: ⟦*1*<*b*;*2*^*x*=*2*∗(*b*::*nat*)⟧ ⟹ *2 dvd b*
⟨*proof*⟩

**lemma** *exp-prod2*: ⟦*1*<*a*; *2*^*x*=*a*∗*2*⟧ ⟹ (*2*::*nat*) *dvd a*
⟨*proof*⟩

**lemma** *odd-mul-odd*: ⟦~(*2*::*nat*) *dvd p*; ~*2 dvd q*⟧ ⟹ ~*2 dvd p*∗*q*
⟨*proof*⟩

**lemma** *prime-equal*: **fixes** *p*::*nat* **shows** ⟦*prime p*; *prime q*; *2*^*x*=*p*∗*q*⟧ ⟹ (*p*=*q*)
⟨*proof*⟩

**lemma** *nat-to-bv-length-bv-to-nat*:
*length xs* = *n* ⟹ *xs* ≠ [] ⟹ *nat-to-bv-length* (*bv-to-nat xs*) *n* = *xs*
⟨*proof*⟩

**end**

# 10 EMSA-PSS encoding and decoding operation

**theory** *EMSAPSS*
**imports** *SHA1 Wordarith*

**begin**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**definition** *emsapss-encode-help7*:: *bv* ⇒ *bv* ⇒ *nat* ⇒ *bv*
**where** *emsapss-encode-help7 maskedDB H emBits* =
*emsapss-encode-help8* (*maskedDB-zero maskedDB emBits*) *H*

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

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

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

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

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

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

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

**definition** *emsapss-decode-help11*:: $bv \Rightarrow bv \Rightarrow bool$
**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* ≠ *bv-prepend* ( (*roundup emBits 8*)∗*8* − (*length mHash*) − *sLen* − *16*) **0** []) ∨ (*take 8* ( *drop* ((*roundup emBits 8*)∗*8* − (*length mHash*) − *sLen* − *16* ) *DB-zero* ) ≠ [*Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *One*])
*then False*
*else emsapss-decode-help8 mHash DB-zero H*)

**definition** *emsapss-decode-help6*:: $bv \Rightarrow bv \Rightarrow bv \Rightarrow nat \Rightarrow bool$
**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* ≠ *bv-prepend* ((*roundup emBits 8*)∗*8* − *emBits*) **0** []
*then False*
*else emsapss-decode-help5 mHash maskedDB* (*MGF H* ((*roundup emBits 8*)∗*8* − (*length mHash*) − *8*)) *H emBits*)

**definition** *emsapss-decode-help3*:: $bv \Rightarrow bv \Rightarrow nat \Rightarrow bool$
**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* ≠ *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* $\leq$ *length M* $\vee$ *2^32*∗*160*<*emBits*)
*then False*
*else emsapss-decode-help1* (*sha1 M*) *EM emBits*)

**lemma** *roundup-positiv*: *0* < *emBits* $\Longrightarrow$ *0* < *roundup emBits 160*
⟨*proof*⟩

**lemma** *roundup-ge-emBits*: *0* < *emBits* $\Longrightarrow$ *0* < *x* $\Longrightarrow$ *emBits* $\leq$ (*roundup emBits x*) ∗ *x*
⟨*proof*⟩

**lemma** *roundup-ge-0*: *0* < *emBits* $\Longrightarrow$ *0* < *x* $\Longrightarrow$ *0* $\leq$ *roundup emBits x* ∗ *x* − *emBits*
⟨*proof*⟩

**lemma** *roundup-le-7*: *0* < *emBits* $\Longrightarrow$ *roundup emBits 8* ∗ *8* − *emBits* $\leq$ *7*
⟨*proof*⟩

**lemma** *roundup-nat-ge-8-help*:
*length* (*sha1 M*) + *sLen* + *16* $\leq$ *emBits* $\Longrightarrow$ *8* $\leq$ *roundup emBits 8* ∗ *8* − (*length* (*sha1 M*) + *8*)
⟨*proof*⟩

**lemma** *roundup-nat-ge-8*:
*length* (*sha1 M*) + *sLen* + *16* $\leq$ *emBits* $\Longrightarrow$ *8* $\leq$ *roundup emBits 8* ∗ *8* − (*length* (*sha1 M*) + *8*)
⟨*proof*⟩

**lemma** *roundup-le-ub*:
⟦ *176* + *sLen* $\leq$ *emBits*; *emBits* $\leq$ *2^32* ∗ *160*⟧ $\Longrightarrow$ (*roundup emBits 8*) ∗ *8* − *168* $\leq$ *2^32* ∗ *160*
⟨*proof*⟩

**lemma** *modify-roundup-ge1*: ⟦*8* $\leq$ *roundup emBits 8* ∗ *8* − *168*⟧ $\Longrightarrow$ *176* $\leq$ *roundup emBits 8* ∗ *8*
⟨*proof*⟩

**lemma** *modify-roundup-ge2*: ⟦ *176* $\leq$ *roundup emBits 8* ∗ *8*⟧ $\Longrightarrow$ *21* < *roundup emBits 8*

⟨*proof*⟩

**lemma** *roundup-help1*: $[\![\ 0 < roundup\ l\ 160]\!] \Longrightarrow (roundup\ l\ 160 - 1) + 1 = (roundup\ l\ 160)$
⟨*proof*⟩

**lemma** *roundup-help1-new*: $[\![\ 0 < l]\!] \Longrightarrow (roundup\ l\ 160 - 1) + 1 = (roundup\ l\ 160)$
⟨*proof*⟩

**lemma** *roundup-help2*: $[\![176 + sLen \leq emBits]\!] \Longrightarrow roundup\ emBits\ 8 * 8 - emBits <= \ roundup\ emBits\ 8 * 8 - 160 - sLen - 16$
⟨*proof*⟩

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

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

**lemma** *length-bv-prepend-drop*: $a <= length\ xs \longrightarrow length\ (bv\text{-}prepend\ a\ b\ (drop\ a\ xs)) = length\ xs$
⟨*proof*⟩

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

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

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

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

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

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

**lemma** *drop-bv-prepend-help4*: $[\![x \leq y]\!] \Longrightarrow drop\ x\ (bv\text{-}prepend\ y\ b\ []) = bv\text{-}prepend\ (y-x)\ b\ []$
⟨*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-bvxor*: *length a* = *length b* ⟹ *length* (*bvxor 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* ⟹ *length* (*MGF1 Z n l*) = *l*
⟨*proof*⟩

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

**lemma** *solve-length-generate-DB*:
⟦ *0* < *emBits*; *length* (*sha1 M*) + *sLen* + *16* ≤ *emBits*⟧
⟹ *length* (*generate-DB* (*generate-PS emBits* (*length* (*sha1 x*)) )) = (*roundup emBits 8*) ∗ *8* − *168*
⟨*proof*⟩

**lemma** *length-maskedDB-zero*:
⟦ *roundup emBits 8* ∗ *8* − *emBits* ≤ *length maskedDB*⟧
⟹ *length* (*maskedDB-zero maskedDB emBits*) = *length maskedDB*
⟨*proof*⟩

**lemma** *take-equal-bv-prepend*:
⟦ *176* + *sLen* ≤ *emBits*; *roundup emBits 8* ∗ *8* − *emBits* ≤ *7*⟧
⟹ *take* (*roundup emBits 8* ∗ *8* − *length* (*sha1 M*) − *sLen* − *16*) (*maskedDB-zero* (*generate-DB* (*generate-PS emBits 160*)) *emBits*) =
*bv-prepend* (*roundup emBits 8* ∗ *8* − *length* (*sha1 M*) − *sLen* − *16*) **0** []
⟨*proof*⟩

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

**lemma** *equal-zero*:
$176 + sLen \leq emBits \Longrightarrow roundup\ emBits\ 8 * 8 - emBits \leq roundup\ emBits\ 8 * 8 - (176 + sLen)$
$\Longrightarrow 0 = roundup\ emBits\ 8 * 8 - emBits - (roundup\ emBits\ 8 * 8 - (176 + sLen))$
⟨*proof*⟩

**lemma** *get-salt*: $[\![ 176 + sLen \leq emBits; roundup\ emBits\ 8 * 8 - emBits \leq 7]\!] \Longrightarrow (generate\text{-}salt\ (maskedDB\text{-}zero\ (generate\text{-}DB\ (generate\text{-}PS\ emBits\ 160))\ emBits)) = salt$
⟨*proof*⟩

**lemma** *generate-maskedDB-elim*: $[\![roundup\ emBits\ 8 * 8 - emBits \leq length\ x; (\ roundup\ emBits\ 8) * 8 - (length\ (sha1\ M)) - 8 = length\ (maskedDB\text{-}zero\ x\ emBits)]\!] \Longrightarrow generate\text{-}maskedDB\ (maskedDB\text{-}zero\ x\ emBits\ @\ y\ @\ z)\ emBits\ (length(sha1\ M)) = maskedDB\text{-}zero\ x\ emBits$
⟨*proof*⟩

**lemma** *generate-H-elim*: $[\![\ roundup\ emBits\ 8 * 8 - emBits \leq length\ x; length\ (maskedDB\text{-}zero\ x\ emBits) =\ (roundup\ emBits\ 8) * 8 - 168; length\ y = 160]\!] \Longrightarrow generate\text{-}H\ (maskedDB\text{-}zero\ x\ emBits\ @\ y\ @\ z)\ emBits\ 160 = y$
⟨*proof*⟩

**lemma** *length-bv-prepend-drop-special*: $[|roundup\ emBits\ 8 * 8 - emBits <= roundup\ emBits\ 8 * 8 - (176 + sLen); length\ (generate\text{-}PS\ emBits\ 160) = roundup\ emBits\ 8 * 8 - (176 + sLen)|] ==> length\ (\ bv\text{-}prepend\ (roundup\ emBits\ 8 * 8 - emBits)\ \mathbf{0}\ (drop\ (roundup\ emBits\ 8 * 8 - emBits)\ (generate\text{-}PS\ emBits\ 160))) = length\ (generate\text{-}PS\ emBits\ 160)$
⟨*proof*⟩

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

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

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

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

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

**lemma** *verify*: $\llbracket(emsapss\text{-}encode\ M\ emBits) \neq [];\ EM{=}(emsapss\text{-}encode\ M\ emBits)\rrbracket \Longrightarrow emsapss\text{-}decode\ M\ EM\ emBits = True$
⟨*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** $rsapss\text{-}sign\text{-}help1 :: nat \Rightarrow nat \Rightarrow nat \Rightarrow bv$
**where** $rsapss\text{-}sign\text{-}help1\ em\text{-}nat\ e\ n =$
$nat\text{-}to\text{-}bv\text{-}length\ (rsa\text{-}crypt\ em\text{-}nat\ e\ n)\ (length\ (nat\text{-}to\text{-}bv\ n))$

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

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

**lemma** *length-emsapss-encode*:
$emsapss\text{-}encode\ m\ x \neq [] \Longrightarrow length\ (emsapss\text{-}encode\ m\ x) = roundup\ x\ 8 * 8$
⟨*proof*⟩

**lemma** *bv-to-nat-emsapss-encode-le*: $emsapss\text{-}encode\ m\ x \neq [] \Longrightarrow bv\text{-}to\text{-}nat\ (emsapss\text{-}encode\ m\ x) < 2\hat{}(roundup\ x\ 8 * 8)$
⟨*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*
(*bvxor*
(*generate-DB*
(*generate-PS* (*length* (*nat-to-bv* ($p * q$)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))
(*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length*
(*generate-DB*
(*generate-PS* (*length* (*nat-to-bv* ($p * q$)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))))) + *168*
⟨*proof*⟩

**lemma** *MGFLen-helper*: *MGF z l* ~= [] ⟹ *l* <= *2*^*32*∗(*length* (*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*))))))) ~= []
**and** *len*: *length* (*sha1 M*) + *sLen* + *16* ≤ (*length* (*nat-to-bv* ($p * q$))) − *Suc 0*
**shows** *length*
(
(*bvxor*
(*generate-DB*
(*generate-PS* (*length* (*nat-to-bv* ($p * q$)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))
(*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length*
(*generate-DB*
(*generate-PS* (*length* (*nat-to-bv* ($p * q$)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))))
) = (*roundup* (*length* (*nat-to-bv* ($p * q$)) − *Suc 0*) *8*) ∗ *8* − *168*

⟨*proof*⟩

**lemma** *emBits-roundup-cancel*: *emBits mod 8* ~= *0* ⟹ (*roundup emBits 8*)∗*8* − *emBits* = *8*−(*emBits mod 8*)
⟨*proof*⟩

**lemma** *emBits-roundup-cancel2*: *emBits mod 8* ~=*0* ⟹ (*roundup emBits 8*) ∗ *8* − (*8*−(*emBits mod 8*)) = *emBits*
⟨*proof*⟩

**lemma** *length-bound*: ⟦*emBits mod 8* ~=*0*; *8* <= (*length maskedDB*)⟧ ⟹ *length* (*remzero* ((*maskedDB-zero maskedDB emBits*)@*a*@*b*)) <= *length* (*maskedDB*@*a*@*b*) − (*8*−(*emBits mod 8*))
⟨*proof*⟩

**lemma** *length-bound2*: *8*<=*length* ( (*bvxor*
  (*generate-DB*
  (*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
  (*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))
  (*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
  (*length*
  (*generate-DB*
  (*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
  (*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))))
⟨*proof*⟩

**lemma** *length-helper*: **assumes** *p*: *prime p* **and** *q*: *prime q* **and** *x*: (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *mod 8* ~= *0* **and** *mgf*: (*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
  (*length*
  (*generate-DB*
  (*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
  (*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))) ~= []
  **and** *len*: *length* (*sha1 M*) + *sLen* + *16* ≤ (*length* (*nat-to-bv* (*p* ∗ *q*))) − *Suc 0*
  **shows** *length*
  (*remzero*
  (*maskedDB-zero*
  (*bvxor*
  (*generate-DB*
  (*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
  (*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))
  (*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
  (*length*
  (*generate-DB*
  (*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
  (*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))))
  (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) @
  *sha1* (*generate-M′* (*sha1 m*) *salt*) @ *BC*))
  < *length* (*nat-to-bv* (*p* ∗ *q*))

$\langle proof \rangle$

**lemma** *length-emsapss-smaller-pq*: $[\![prime\ p;\ prime\ q;\ emsapss\text{-}encode\ m\ (length\ (nat\text{-}to\text{-}bv\ (p * q)) - Suc\ 0) \neq [];\ (length\ (nat\text{-}to\text{-}bv\ (p * q)) - Suc\ 0)\ mod\ 8\ {}^\sim\!=\ 0]\!] \Longrightarrow length\ (remzero\ (emsapss\text{-}encode\ m\ (length\ (nat\text{-}to\text{-}bv\ (p * q)) - Suc\ 0))) < length\ (nat\text{-}to\text{-}bv\ (p*q))$
$\langle proof \rangle$

**lemma** *bv-to-nat-emsapss-smaller-pq*: **assumes** $a$: $prime\ p$ **and** $b$: $prime\ q$ **and** $pneq$: $p\ {}^\sim\!=\ q$ **and** $c$: $emsapss\text{-}encode\ m\ (length\ (nat\text{-}to\text{-}bv\ (p * q)) - Suc\ 0) \neq []$
**shows** $bv\text{-}to\text{-}nat\ (emsapss\text{-}encode\ m\ (length\ (nat\text{-}to\text{-}bv\ (p * q)) - Suc\ 0)) < p*q$
$\langle proof \rangle$

**lemma** *rsa-pss-verify*: $[\![\ prime\ p;\ prime\ q;\ p \neq q;\ n = p*q;\ e*d\ mod\ ((pred\ p)*(pred\ q)) = 1;\ rsapss\text{-}sign\ m\ e\ n \neq [];\ s = rsapss\text{-}sign\ m\ e\ n\ ]\!] \Longrightarrow rsapss\text{-}verify\ m\ s\ d\ n = True$
$\langle proof \rangle$

**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.