Formal Proof for the Correctness of RSA-PSS ⋆

Christina Lindenberg, Kai Wirt, and Johannes Buchmann

Darmstadt University of Technology
Department of Computer Science
Darmstadt, Germany
{lindenberg,wirt}@informatik.tu-darmstadt.de
buchmann@cdc.informatik.tu-darmstadt.de

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. This paper is one step to provide a tool box allowing the use of formal methods in every aspect of cryptography. In this paper we give a formal specification of the RSA probabilistic signature scheme (RSA-PSS) [4] which is used as algorithm for digital signatures in the PKCS #1 v2.1 standard [7]. Additionally we show the correctness of RSA-PSS. This includes the correctness of RSA, the formal treatment of SHA-1 and the correctness of the PSS encoding method. Moreover we present a proof of concept for the feasibility of verification techniques to a standard signature algorithm.

Keywords: cryptography, specification, verification, digital signature

1 Motivation

Todays software often contains many errors which are not discovered during the development. Although erroneous software is mostly only annoying, bugs may lead to severe security issues as well. Moreover bugs even can have huge impacts if they appear in software used for critical applications such as controlling software in nuclear power plants. There are various examples of computer related accidents which led to loss of lives like the crash of the Korean Air Lines B747 in Guam 1997 or the Therac-25 radiation-therapy machine which gave patients massive overdoses between 1985 and 1987 [11], [9], [16]. The reason for such poor software is, that not all errors can be found by tests. Even if programs are very intensively tested they may still contain several more or less severe bugs.

A possible solution to this dilemma is the formal verification of software. The goal of the application of formal methods in program verification is to prove the correctness of software, that is to give a mathematical proof that the software fulfills its specification. If a formal proof for the correctness of a program is

⋆ This work was partially funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the authors.

given, there is no need for any tests. Hence, the verified systems are of extreme quality as required in many industrial sectors, such as automotive engineering, security, and medical technology. However to give a formal proof one needs to have a formal specification of the software in question[1].

In this paper we give such a formal specification of the RSA probabilistic signature scheme (RSA-PSS) [4] which is used as algorithm for digital signatures in the PKCS #1 v2.1 standard [7]. For our work we used the Isabelle/HOL theorem prover [13] [10] which is developed at Cambridge University and TU Munich. Simply speaking a theorem prover is a computer assistant for formal proofs.

The major advantage of RSA-PSS over the widely used older PKCS #1 v1.5 standard, which simply uses a padded message digest as input to the signature algorithm, is, that RSA-PSS can be proven secure in the Random Oracle Model [2]. Additionally it does not contain certain critic points of the older standard. Therefore, new signature applications should use the probabilistic signature scheme. Our intention is to provide a basis for a rigorous treatment of RSA-PSS using formal methods. Therefore we present a correctness proof of RSA-PSS. This means we formally show, that a signature can always be verified (i.e. functional correctness). Our work allows one to verify if an actual implementation of RSA-PSS is correct according to the specification. This is not possible using the PKCS document alone. Additionally we see our work as proof of concept in the sense that we show, that it is possible to use formal methods in cryptography. This is not obvious because of the inherent complexity of practical cryptosystems like RSA-PSS. This can be clearly seen at our herein presented correctness proof for which we had to show theorems on the RSA function, the secure hash algorithm and the probabilistic signature scheme, or in other words, to show a certain property of a standard cryptographic method one has to reason about various cryptographic primitives. As far as we know, our work is the first attempt to use formal methods to verify properties of complete standard cryptographic signature schemes.

While formal verification of programs becomes more and more important, formal verification of cryptographic primitives is still in the fledgling stages. The need for a fundamental set of formal theories covering a broad range of methods from cryptography arises because of the demand for continuity in formal proofs of security relevant applications. The presented framework is one step on the way to the construction of a tool box allowing the application of formal methods to cryptography. For related research we refer to the publications of Backes and Pfitzmann [1], Boyer and Moore [5], and Dolev-Yao [6].

The paper is organized as follows: In section 2 we present the RSA-PSS signature scheme and give our formal specification. The complete correctness proof is the topic of section 3. We conclude in section 4. The complete formal specification and the proof scripts for Isabelle/HOL are contained in an appendix.

[1] There exist automatic tools to translate software source code into the language of a theorem proving environment. In this environment it is possible to show the equivalence of the translated source code and the formal specification.

2 The Digital Signature Scheme RSA-PSS and its Formal Specification

In this section we give a short survey of RSA and RSA-PSS. We also present our formal specification of RSA and PSS. For RSA we geared to [5]. The SHA-1 specification is directly derived from [8] and the PSS encoding method was specified according to [7]. Since the PSS encoding method is generic in the sense that the signature algorithm and the hash function used are not specified our RSA-PSS theory is combining the different parts mentioned above.

2.1 Introduction

One important component of secure data communication is a digital signature. It assures authentication, authorization and non-repudiation. The digital signature we consider here is RSA-PSS. RSA-PSS is a signature scheme with appendix. Such a scheme consists of a signature-generation operation and a signature-verification operation. A signature is produced for a message with the signers private key. To verify if a signature is valid the verifier needs the signature, the message for which the signature was produced and the public key of the signer. Signature schemes with appendix are distinguished from signature schemes with message recovery, see [12].

2.2 Public-Key Signatures

A public-key signature scheme consists of a signing procedure and a verification procedure. For a message m the signer creates a signature s with his private key. Then he sends the pair (m, s) to a person who wants to verify his signature. The verifier uses the public key of the signer to check, if the signature s is a valid signature for the message m. One possible public-key signature scheme is the RSA signature scheme. Instead of decrypting a message m, the signer uses his private key to generate a signature s of the message m. A verifier can now use the public key of the signer to check the signature. If the decryption of the signature s is equal to m, then s is a valid signature of the message m.

2.3 Asymmetric cryptographic system - RSA

In an asymmetric cryptographic system every user has a public key and a corresponding private key. The public key is available for everyone, the private key has to be kept secret. Of course it is hard to derive the private key from the public key. With an encryption algorithm and a public key every user can encrypt a message. The decryption of the message can only be done by the user who knows the corresponding private key. Mathematically seen, a public key system assumes the existence of trapdoor one-way functions.

The most common public key cryptosystem is RSA which was invented by R. Rivest, A. Shamir and L. Adleman [14] in 1978. Since then the algorithm has been analyzed by many experts from all over the world but the security has never been disproved neither proved. The great advantage of this cryptosystem is the simplicity of understanding and its application. The security of RSA is assumed on the intractability of the integer-factorization problem. We will now give a short sketch of RSA.

Let p and q be random prime numbers with $p \neq q$. Compute $n = pq$. Select a random number e, with $1 < e < (p-1)(q-1)$, such that $\gcd(e, (p-1)(q-1)) = 1$. Furthermore compute the unique integer d, $1 < d < (p-1)(q-1)$, such that $ed \equiv 1 \bmod (p-1)(q-1)$. The public key is (n, e) and the private key is d. The integer e is called the encryption exponent, d the decryption exponent and n the modulus. The encryption of a message m, $0 \leq m < n$, is computed by $c = m^e \bmod n$, where c is called the cipher text of the message m. To recover the message m from the cipher text c, compute $m = c^d \bmod n$. For the correctness proof see [14], [5].

For the specification of our RSA function we use the same "binary method" as [5] (fast exponentiation).

$$m^e \bmod n = \begin{cases} (m^{e/2})^2 \bmod n & : \text{ if } e \text{ is even} \\ m(m^{e/2})^2 \bmod n & : \text{ if } e \text{ is odd} \end{cases}$$

Additionally we formally show, that our method which performs the fast exponentiation indeed calculates the ordinary exponentiation. This can be done by simple induction on the exponent.

2.4 The Secure Hash Algorithm

In the encoding process of PSS a hash function is required. A hash function takes an input of variable length and maps it to a so-called message digest of fixed size. A cryptographic hash function has to satisfy three security properties. First it has to be *collision resistant*, that is, it must be computationally infeasible to find any two messages which lead to the same hash value. Second, given a hash value, it must be infeasible to find a message which hashes to that value (*first preimage resistance*) and third it has to be difficult given one message to find another message such that both hash to the same value (*second preimage resistance*).

In our work we used the Secure Hash Algorithm (SHA-1) [8]. SHA-1 was widely believed to have the above mentioned security properties. However recently a technical report by Wang, Yin and Yu [15] was published which claims to break the collision resistance property. Since the hash function is exchangeable in the PSS construction the concrete internals of SHA-1 are irrelevant for the correctness proof. However they are necessary for the formal specification, i.e. if one wishes to verify a software implementation. We stress, that using our techniques it is possible to exchange the hash function in the formal proof as a response to the above mentioned attack but we decided to hold on to SHA-1 because of the fact, that it is the most commonly used hash function today.

Our SHA-1 specification is a direct application of the FIPS standard [8]. The main problem on the realization in a formal proof system is, that SHA-1 doesn't have an easy mathematical structure but operates on the bit level. Therefore somehow the concept of bit vectors has to be added to the proof system. One has to add support to the proof system for hexadecimal numbers and methods to convert these to bit vectors thus providing an easy way to model constants used in the description of SHA-1. Additionally one has to define logical and, inclusive and exclusive or operations on bit vectors as well as the circular shift. Additionally we need a way to break bit vectors into components, we need an addition modulo 2^{32} and a way to create arbitrary long bit vectors which are completely 0.

Using this extensions it becomes possible to define the message padding for SHA-1, which is given by appending 0 and the 64-bit representation of the original message length such that the length of the padded message is a multiple of 512 bits.

The SHA-1 theory contains the actual specification for SHA-1. This specification is split into various functions similar to the description in the FIPS document.

2.5 The PSS encoding method

The PSS encoding method was developed by Bellare and Rogaway in [3] and [4]. A variant of this scheme is described in the PKCS v 1.5 [7] standard document. Our specification is a direct application of this standard. Our specification makes use of the length of the used hash function. We have implemented the SHA-1 function since it is the state of the art hash. However it is possible to exchange the used hash function without major changes on the rest of the specification or our proofs. PSS essentialy uses two functions. The first one generates the encoded fingerprint of a given message. The other one takes the encoded fingerprint along with a message and checks wether the encoding of the fingerprint is correct for the message.

EMSA-PSS-Encoding Operation. The PSS encoding method is described in algorithm 1 and figure 1. Our formal specification is a direct implementation of this algorithm. In our specification *salt* is the empty string, which has the length 0. That is a typical *salt* length according to [7]. As hash function we use sha1, which is specified in 2.4.

EMSA-PSS-Decoding Operation. If a signature is a valid signature of a message, it can be verified by algorithm 2.

Mask Generation Function. Mask generation functions take an arbitrary value x and the desired length l for the output and compute a hash value of length l. Mask generation functions are deterministic, i.e. the output is completely determined by the input value. Also the output should be pseudo-random this means that given one part of the output and not the input it should be infeasible

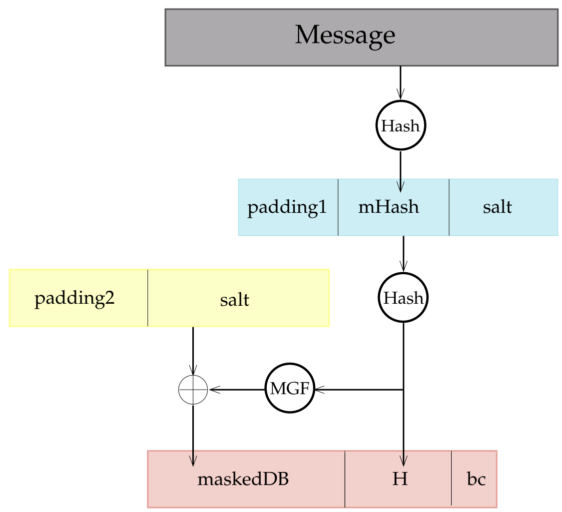

Fig. 1. encoding operation

Algorithm 1 EMSA-PSS-Encode

Input: message m to be encoded, an octet string
maximal bit length *emBits* of the output message, at least $8hLen + 8sLen + 9$
Options: Hash function ($hLen$ is the length in octets of the hash function output)
$sLen$ intended length in octets of the salt
Output: encoded message em, an octet string of length $emLen = \lceil emBits/8 \rceil$

if length of m is greater than input limitation for the hash function output "error"
$mHash \leftarrow \text{Hash}(m)$
if $emLen < hLen + sLen + 2$ output "error"
generate a random octet string *salt* of length $sLen$
$m' \leftarrow$ (0x)00 00 00 00 00 00 00 00 $\|$ $mHash$ $\|$ $salt$
$H \leftarrow \text{Hash}(m')$
generate a octet string PS consisting of $emLen - sLen - hLen - 2$ zero octets, the length may be 0
$DB \leftarrow PS \;\|\; \text{0x01} \;\|\; salt$
$dbMask \leftarrow \text{MGF}(H, emLen - hLen - 1)$
$maskedDB \leftarrow DB \oplus dbMask$
set the leftmost $8emLen - emBits$ bits of the leftmost octet in *maskedDB* to zero
$em \leftarrow maskedDB \;\|\; H \;\|\; \text{0xBC}$

---

to get some information about another part of the output. Mask generation functions can be build from hash functions (e.g. SHA-1). The security of RSA-PSS depends on the randomness of the mask generation function and this again on the randomness of the used hash function. We used the mask generation function described in Algorithm 3.

Algorithm 2 EMSA-PSS-Decoding

Input: message m to be verified, an octet string
encoded message em, an octet string of length $emLen = \lceil emBits/8 \rceil$
maximal bit length $emBits$ of the output message, at least $8hLen + 8sLen + 9$
Options: Hash function ($hLen$ is the length in octets of the hash function output)
$sLen$ intended length in octets of the salt
Output: "valid" or "invalid"

```
if length of m is greater than the input limitation for the hash function output
  "invalid"
mHash ← Hash(m)
if emLen < hLen + sLen + 2 output "invalid"
if the rightmost octet of em does not have hexadecimal value 0xBC, output "in-
  valid"
maskedDB ← the leftmost emLen − hLen − 1 octets of em and
H ← the next hLen octets
if the 8emLen − emBits bits of the leftmost octet in maskedDB are not all equal
  to zero, output "invalid"
dbMask ← MGF(H, emLen − hLen − 1)
DB ← maskedDB ⊕ dbMask
set the leftmost 8emLen − emBits bits of the leftmost octet in DB to zero
if the emLen − hLen − sLen − 2 leftmost octets of DB are not zero or if the octet at
  position emLen − hLen − sLen − 1 does not have hexadecimal value 0x01, output
  "invalid"
salt ← the last sLen octets of DB
m′ ← (0x)00 00 00 00 00 00 00 00 ‖ mHash ‖ salt
H′ ← Hash(m′)
if H = H′ then output "valid", otherwise output "invalid"
```

Algorithm 3 MGF1

Input: $mgfSeed$: seed from which the mask is generated, an octet string
$maskLen$: intended length in octets of the mask, at most $2^{32}hLen$
Output: $mask$: an octet sting of length $maskLen$

```
if maskLen > 2^32 hLen then output "error"
T ⇐ ϵ
for counter = 0 to ⌈maskLen / hLen⌉ − 1 do
  T ⇐ T ‖ Hash(mgfSeed ‖ C), where C is the counter converted to an octet
  string of length 4
end for
mask ⇐ the leading maskLen octets of T
```

2.6 Construction of RSA-PSS

RSA-PSS is the combination of the previously described primitives. RSA-PSS uses the RSA function to sign the PSS encoded data. The verification is achieved by using the public key to "encrypt" the signature which again yields the PSS encoded fingerprint. The fingerprint is then checked for consistency using the above described decoding procedure.

The complete RSA-PSS Signature Scheme consists of the following functions:

RSASP1 $((n, d), m)$	The RSA signature-primitive computes for the input private key (n, d) and a message m, $0 \leq m < n$ the signature $s = m^d \bmod n$.
RSAVP1 $((n, e), s)$	The RSA verification-primitive computes for the input public key (n, e) and the signature s the corresponding message $m = s^e \bmod n$.
Hash(m)	A hash function (e.g. SHA-1) which computes for a message m with arbitrary length a hash value of fixed length.

We also define two functions (emsapss_encode m *emBits*), which encodes the fingerprint of a message m in a bit string of maximum length *emBits* and (emsapss_decode m *em* *emBits*), which decides for a message m, an encoded fingerprint *em* and the maximum length *emBits* of *em*, if *em* is a valid encoding of m. The following algorithms are specified in [7], see there for a full description.

Signature-Generation Operation. In algorithm 4 we describe the generation of a RSA-PSS signature. This algorithm is the basis for our formal specification.

Algorithm 4 RSA-PSS signature generation

Input: signer's RSA private key (n, d)
message m to be signed, an octet string
Output: signature s, an octet string
1: $modBits \leftarrow$ bit length of the RSA modulus n
2: $em \leftarrow$ emsapss_encode$(m, modBits - 1)$
3: $s \leftarrow$ RSASP1 $((n, d), em)$

Signature-Verification Operation. The verification of a RSA-PSS signature is done in two steps. First, the RSAVP1 function is applied to the signature to get the encoded message. After this, the emsapss_decode operation is applied to the message and the encoded message to determine wether they are consistent, see algorithm 5.

3 Correctness Proof

It becomes very difficult and complex to show the correctness directly for the complete RSA-PSS encoding method. However it is possible to split this task into several smaller parts which can then be verified much easier. Our approach is to first give a proof for the pure RSA function, namely $(m^e)^d \bmod n = m$. Secondly we prove: (emsapss_decode m (emsapss_encode m *emBits*) *emBits*) = True. The last step of the complete proof is to combine the individual parts. Although this step seems simple at first sight, there are various obstacles which we will point out in the corresponding subsection.

Algorithm 5 RSA-PSS signature verification

Input: signer's RSA private key (n, d)
message m whose signature is to be verified, an octet string
signature s to be verified, an octet string

Output: valid or invalid signature

1: $modBits \longleftarrow$ bit length of the RSA modulus n
2: $em \longleftarrow$ RSAVP1 $((n, e), s)$
3: $Result \longleftarrow$ emsapss_decode(m, em, $modBits - 1$)
4: if $Result$ = "valid" then output "valid signature" otherwise "invalid signature"

3.1 Correctness of RSA

The correctness proof of the RSA function makes use of Fermat's little theorem. Due to space limitations we omit the formal proof of this theorem at this point and state simply the theorem itself which is then used in the further proof.

lemma *fermat*: $[\![p \in prime;\ m\ mod\ p \neq 0]\!] \Longrightarrow m\hat{\ }(p-(1::nat))\ mod\ p = 1$

The correctness statement of RSA in Isabelle notation is:

lemma *cryptinverts*:
$[\![p \in prime;\ q \in prime;\ p \neq q;\ n = p*q;\ m < n;$
$e*d\ mod\ ((pred\ p)*(pred\ q)) = 1]\!] \Longrightarrow$
$rsa\text{-}crypt\ (rsa\text{-}crypt\ (m,e,n),\ d\ ,\ n) = m$

which basically says, that if one uses the private key to encrypt (i.e. sign) a message m and afterwards uses the public key to encrypt (i.e. verify) the result, then one again has m.

Since the RSA correctness proof is mainly number theoretic it can be easily shown in a theorem proving environment. The main tools one needs are lemmata on modular arithmetic and on properties of primes. Fermat's little theorem is then established using some theorems on permutations of natural numbers.

Our proof closely abides by the prior work of Boyer-Moore [5] however we were not able to translate it one to one to Isabelle due to differences in the basic libraries of the theorem provers. Therefore we had to extend Boyer and Moores proof in order to adapt it to Isabelle.

3.2 Length of SHA-1

In this section we present the proof of the length of SHA-1 which is required to show the correctness of the RSA-PSS signature scheme. In principle it would also be possible to define an abstract hash function and give the correctness proof for every such function, which has a certain minimal length. However since we decided to give a specification which can be used to verify actual implementations we specified the SHA-1 hash function and have to give a proof for the length of this certain function. The concrete proof is quite easy since the length of SHA-1 is the addition of five 32-bit blocks as can be seen from the definition of SHA-1.

3.3 Correctness of the PSS-Encoding Method

In this section we give the formal proof, that for a message m, and the encoded message em of m, with $em \neq []$ the function emsapss_decode returns True.

The proof basically is established by looking at a encoded message showing, that this message has a certain format. The first step is to show that the least significant eight bits of the encoded message are 0xBC. We then have to show, that the leftmost bits are equal to zero. This is an important property for the complete proof, because it ensures, that the encoded message when interpreted as natural number is smaller than the RSA modulus, which allows us to apply the RSA correctness proof.

Another important tool is to show, that the application of two times bitwise xor with the same mask leaves a bitvector unchanged. Therefore it is possible to cancel out the effect of the masking operation. This yields the padding2 string which can be checked for correctness and the salt, which can then be used together with the padding1 string to verify the actual fingerprint.

The rest of this proof can be shown by straightforward substitutions and the application of the above mentioned theorems. The main problems here are of technical nature. Due to the complexity of the expressions it becomes complicated to keep the track of the proof. Our research indicated, that theorem provers which are used to verify cryptographic algorithms should somehow ease the reasoning with complex expressions.

3.4 Combination of the single proofs

We now show that a RSA-PSS signature s for a message m can always be verified with our RSA-PSS specification from section 2.6. Formally we prove the following

lemma *rsa-pss-verify*:
  ⟦*p* ∈ *prime*; *q* ∈ *prime*; *p* ≠ *q*; *n* = *p*∗*q*;
   e∗*d mod* ((*pred p*)∗(*pred q*)) = *1*; *rsapss-sign m e n* ≠ [];
    s = *rsapss-sign m e n*⟧
  ⟹ *rsapss-verify m s d n* = *True*.

In the following we use $|\cdot|$ to denote the length of the bitvector representing the number $\cdot$.

In order to apply the correctness lemma for RSA which gives us em in the verification step, we have to show that $em < n$. This indeed is the major obstacle in combining the single proofs described above.

To show that $em < n$ we use the preconditions $p, q \in prime$, $p \neq q$ and $n = p \cdot q$. Our approach is to distinguish wether em starts with 0 or 1-bits. The first case is easy because we can show that preceding zeroes do not change the value of a bit vector. In other words if we denote with $em^\star$ the value of em with the leading zeros removed we can show that $em^\star = em$ and $|em^\star| < |n|$. Since we have $|em^\star| < |n| \Rightarrow em^\star < n$ we have shown the first case (Note, that n does always start with a 1-bit because of our specification).

In the second case we can show that $|em| = |p \cdot q| - 1$ and $0 < p \cdot q - 1$. Additionaly we have $0 < p \cdot q - 1 \Rightarrow 2^{|p \cdot q| - 1} \leq p \cdot q$. Thus all that remains to show is that $2^{|p \cdot q| - 1} \neq p \cdot q$. This can be done by showing that the only possible product of two prime numbers which is a power of 2 is $2 \cdot 2$. This however is not allowed since we have the precondition that $p \neq q$.

Another problem is again the inherent complexity of the occuring expressions. In this step one has to switch between natural numbers and the bitvector description of the numbers which always introduces one layer of indirection. This issue is typical for the verification of cryptographic algorithms since they mix operations in different fields like $GF(2)$ and $\mathbb{Z}_n$ in order to prevent attacks. One possible solution is to show theorems which allow to cancel out the transformation functions. However care must be taken with the order of the application of the functions since for example the conversion from bitvector to natural and back removes leading zeros from the bitvector description.

4 Conclusion

In this paper we presented a formal specification of the RSA probabilistic signature scheme. Moreover we verified the functional correctness property of RSA-PSS using formal methods. Further research in this area is very important because of the lack of formal tools which can be used to verify certain cryptographic algorithms. Our aim is to formalize the paper and pencil security proof given for RSA-PSS. On this way there are many interesting topics which have to be done first. One very important point to mention is to formally describe the random oracle model. Also there is not much theory on how to analyse programs with respect to their time and space complexity which would allow to model adversaries for a theorem proving environment.

Using the herein presented specification of RSA-PSS it becomes possible to verify the correctness of actual implementations of RSA-PSS. Up until now, this could only be done by using so called test vectors, which is an indication of the correctness but it constitutes no proof. Although we know, that our work is only one step on a complete formal treatment of RSA-PSS, we feel that the presented proofs encourage further research in this area as they show, that it is possible to verify complex cryptographic protocols like RSA-PSS.

As a closing remark we stress, that formal methods are also of great use to understand proofs. Using theorem proving environments one becomes aware of pitfalls which arise during the proof and which often are overlooked, when doing proofs on paper.

References

1. Michael Backes and Birgit Pfitzmann. Symmetric encryption in a simulatable dolev-yao style cryptographic library. In *CSFW*, pages 204–218, 2004.
2. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

3. Mihir Bellare and Phillip Rogaway. The Exact Security of Digital Signatures — How to Sign with RSA and Rabin. *Lecture Notes in Computer Science*, 1070:399–416, 1996.
4. Mihir Bellare and Phillip Rogaway. PSS: Provably Secure Encoding Method for Digital Signatures. *Submission to IEEE P1363*, 1998.
5. Robert S. Boyer and J. Strother Moore. Proof checking the rsa public key encryption algorithm. Technical Report 33, Institute for Computing Science and Computer Applications, University of Texas, 1982.
6. Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
7. PKCS Editor. PKCS#1 v2.1: RSA Cryptography Standard. Technical report, RSA Laboratories, 2002.
8. Federal Information Processing Standards. Secure hash standard. Technical Report FIPS 180-2, National Institute of Standards and Technology, 2002.
9. Thomas Huckle. Collection of software bugs. http://www5.in.tum.de/˜huckle/bugse.html, 2004.
10. Development website of isabelle at the tu munich. http://isabelle.in.tum.de.
11. Peter B. Ladkin. Computer related incidents with commercial aircraft. http://www.rvs.uni-bielefeld.de/publications/Incidents/.
12. Alfred J. Menezes, Paul van Oorschot, and Scott Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
13. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
14. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.
15. Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Collision search attacks on sha1. http://theory.csail.mit.edu/˜yiqun/shanote.pdf, 2005.
16. Article computer bug. http://en.wikipedia.org/wiki/Computer_bug.

A Formal Specification of RSA

theory *Crypt* = *Mod*:

constdefs
even :: *nat* ⇒ *bool*
even n == *2 dvd n*

consts
rsa-crypt :: *nat* × *nat* × *nat* => *nat*

recdef *rsa-crypt measure*(λ(*M*,*e*,*n*).*e*)
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*))ˆ*2 mod n*) *else*
(*M* ∗ ((*rsa-crypt* (*M*, *Suc e div 2*,*n*))ˆ*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*))

```
  by (simp add: even-def dvd-eq-mod-eq-0 mult-commute mult-div-cancel)

theorem cryptcorrect [rule-format]:
  ((n ≠ 0) & (n ≠ 1)) ⟶ (rsa-crypt(M,e,n) = M^e mod n)
  apply (induct-tac M e n rule: rsa-crypt.induct)
  by (auto simp add: power-mult [THEN sym] div-2-times-2 remainderexp
    timesmod1)

end
```

B Fermat's little theorem

```
theory Fermat = Pigeonholeprinciple:

consts
  pred :: nat ⇒ nat
  S :: nat * nat * nat ⇒ nat list

primrec
  pred 0 = 0
  pred (Suc a) = a

recdef S measure(λ(N,M,P).N)
  S (0,M,P) = []
  S (N,M,P) = (((M*N) mod P)#(S ((N−(1::nat)),M,P)))

lemma remaindertimeslist:
  timeslist (S(n,M,p)) mod p = fac n * M^n mod p
  apply (induct-tac n M p  rule: S.induct)
  apply (auto)
  apply (simp add: add-mult-distrib)
  apply (simp add: mult-assoc [THEN sym])
  apply (subst add-mult-distrib [THEN sym])
  apply (subst mult-assoc)
  apply (subst mult-left-commute)
  apply (subst add-mult-distrib2 [THEN sym])
  apply (simp add: mult-assoc)
  apply (subst mult-left-commute)
  apply (simp add: mult-commute)
  apply (subst mod-mult1-eq' [THEN sym])
  apply (drule remainderexplemma)
  by (auto)

lemma sucassoc: (P + P*w) = P * Suc w
  by (auto)

lemma modI [rule-format]: 0 < (x::nat) mod p ⟶ 0 < x
  by (induct-tac x, auto)
```

lemma *delmulmod*: ⟦*0* < *x mod p*;*a* < (*b*::*nat*)⟧ ⟹ *x*∗*a* < *x*∗*b*
by (*simp*, *rule modI*, *simp*)

lemma *swaple* [*rule-format*]:
(*c* < *b*) ⟶ ((*a*::*nat*) ≤ *b* − *c*) ⟶ *c* ≤ *b* − *a*
apply (*induct-tac a*, *auto*)
apply (*subgoal-tac c*~= *b* − *n*, *auto*)
apply (*drule le-neq-implies-less*[*of c*])
apply (*simp*)+
by (*arith*)+

lemma *exchgmin*: ⟦(*a*::*nat*) < *b*;*c* ≤ *a*−*b*⟧ ⟹ *c* ≤ *a* − *a*
by (*auto*)

lemma *sucleI*: *Suc x* ≤ *0* ⟹ *False*
by (*auto*)

lemma *diffI*: ⋀*b*. (*0*::*nat*) = *b* − *b*
by (*auto*)

lemma *alldistincts* [*rule-format*]:
(*p*: *prime*) ⟶ (*m mod p* ≠ *0*) ⟶ (*n2* < *n1*) ⟶ (*n1* < *p*) ⟶
¬(((*m*∗*n1*) *mod p*) *mem* (*S* (*n2*,*m*,*p*)))
apply (*induct-tac rule*: *S*.*induct*)
apply (*auto*)
apply (*drule equalmodstrick2*)
apply (*subgoal-tac M*+*M*∗*w* < *M*∗*n1*)
apply (*auto*)
apply (*drule dvdI*)
apply (*simp only*: *sucassoc diff-mult-distrib2*[*THEN sym*])
apply (*drule primekeyrewrite*, *simp*)
apply (*simp add*: *dvd-eq-mod-eq-0*)
apply (*drule-tac n*=*n1* − *Suc w* **in** *dvd-imp-le*, *simp*)
apply (*rule sucleI*, *subst diffI* [*of n1*])
apply (*rule exchgmin*, *simp*)
apply (*rule swaple*, *auto*)
apply (*subst sucassoc*)
apply (*rule delmulmod*)
by (*auto*)

lemma *alldistincts2* [*rule-format*]:
(*p*: *prime*) ⟶ (*m mod p* ≠ *0*) ⟶ (*n* < *p*) ⟶
alldistinct (*S* (*n*,*m*,*p*))
apply (*induct-tac rule*: *S*.*induct*)
apply (*simp*)+
apply (*subst sucassoc*)
apply (*rule impI*)+
apply (*rule alldistincts*)
by (*auto*)

lemma *notdvdless*: ¬ *a dvd b* ⟹ *0* < (*b*::*nat*) *mod a*
  apply (*rule contrapos-np*, *simp*)
  by (*simp add*: *dvd-eq-mod-eq-0*)

lemma *allnonzerop* [*rule-format*]: (*p*: *prime*) ⟶
 (*m mod p* ≠ *0*) ⟶ (*n* < *p*) ⟶ *allnonzero* (*S* (*n*,*m*,*p*))
  apply (*induct-tac rule*: *S*.*induct*)
  apply (*simp*)+
  apply (*auto*)
  apply (*subst sucassoc*)
  apply (*rule notdvdless*)
  apply (*clarify*)
  apply (*drule primekeyrewrite*)
  apply (*assumption*)
  apply (*simp add*: *dvd-eq-mod-eq-0*)
  apply (*drule-tac n*=*Suc w* **in** *dvd-imp-le*)
  by (*auto*)

lemma *predI* [*rule-format*]: *a* < *p* ⟶ *a* ≤ *pred p*
  apply (*induct-tac p*)
  by (*auto*)

lemma *predd*: *pred p* = *p*−(*1*::*nat*)
  apply (*induct-tac p*)
  by (*auto*)

lemma *alllesseqps* [*rule-format*]:
  p ≠ *0* ⟶ *alllesseq* (*S* (*n*,*m*,*p*)) (*pred p*)
  apply (*induct-tac n m p rule*: *S*.*induct*)
  apply (*auto*)
  by (*simp add*: *predI mod-less-divisor*)

lemma *lengths*: *length* (*S* (*n*,*m*,*p*)) = *n*
  apply (*induct-tac n m p rule*: *S*.*induct*)
  by (*auto*)

lemma *suconeless* [*rule-format*]: *p*: *prime* ⟶ *p* − *1* < *p*
  apply (*induct-tac p*)
  by (*auto simp add*:*prime-def*)

lemma *primenotzero*: *p*: *prime* ⟹ *p*≠*0*
  by (*auto simp add*:*prime-def*)

lemma *onemodprime* [*rule-format*]: *p*:*prime* ⟶ *1 mod p* = (*1*::*nat*)
  apply (*induct-tac p*)
  by (*auto simp add*:*prime-def*)

lemma *fermat*: ⟦*p* ∈ *prime*; *m mod p* ≠ *0*⟧ ⟹ *m*ˆ(*p*−(*1*::*nat*)) *mod p* = *1*
  apply (*frule onemodprime* [*THEN sym*], *simp*)
  apply (*frule-tac n* =*p*− *Suc 0* **in** *primefact*)

```
  apply (drule suconeless, simp)
  apply (erule ssubst)
  back
  apply (rule-tac M = fac (p − Suc 0) in primekeytrick)
  apply (subst remaindertimeslist [of p − Suc 0 m p, THEN sym])
  apply (frule-tac n = p−(1::nat) in alldistincts2, simp)
  apply (rule suconeless, simp)
  apply (frule-tac n = p−(1::nat) in allnonzerop, simp)
  apply (rule suconeless, simp)
  apply (frule primenotzero)
  apply (frule-tac n = p−(1::nat) and m = m and p = p in alllesseqps)
  apply (frule primenotzero)
  apply (simp add: predd)
  apply (insert lengths [of p−Suc 0 m p, THEN sym])
  apply (insert pigeonholeprinciple [of S (p−(Suc 0), m, p)])
  apply (auto)
  apply (drule permtimeslist)
  by (simp add: timeslistpositives)

end
```

C Correctness Proof for RSA

```
theory Cryptinverts = Fermat + Crypt:

lemma cryptinverts-hilf1:
  [[p ∈ prime]] ⟹ (m * m ^(k * pred p)) mod p = m mod p
  apply (case-tac m mod p = 0)
  apply (simp add: mod-mult1-eq')
  apply (simp only: mult-commute [of k pred p] power-mult mod-mult1-eq
    [of m (m^pred p)^k p] remainderexp
    [of m^pred p p k, THEN sym])
  apply (insert fermat [of p m])
  apply (simp add: predd)
  apply (subst sucis)
  apply (subst oneexp)
  apply (subst onemodprime)
  by (auto)

lemma cryptinverts-hilf2:
  [[p ∈ prime]] ⟹ m*(m^(k * (pred p) * (pred q))) mod p = m mod p
  apply (simp add: mult-commute [of k * pred p pred q] mult-assoc
    [THEN sym])
  apply (rule cryptinverts-hilf1 [of p m (pred q) * k])
  by (simp)

lemma cryptinverts-hilf3:
  [[q ∈ prime]] ⟹ m*(m^(k * (pred p) * (pred q))) mod q = m mod q
  apply (simp only: mult-assoc)
```

```
  apply (simp add: mult-commute [of pred p pred q])
  apply (simp only: mult-assoc [THEN sym])
  apply (rule cryptinverts-hilf2)
  by (simp)

lemma cryptinverts-hilf4: ⟦p ∈ prime; q ∈ prime; p ≠ q; m < p*q;
  x mod ((pred p)*(pred q)) = 1⟧ ⟹ m^x mod (p*q) = m
  apply (frule cryptinverts-hilf2 [of p m k q])
  apply (frule cryptinverts-hilf3 [of q m k p])
  apply (frule mod-eqD)
  apply (elim exE)
  apply (rule specializedtoprimes1a)
  by (simp add: cryptinverts-hilf2 cryptinverts-hilf3 mult-assoc
  [THEN sym])+

lemma primmultgreater:
  ⟦ p ∈ prime; q ∈ prime; p ≠ 2; q ≠ 2⟧ ⟹ 2 < p*q
  apply (simp add:prime-def)
  apply (insert mult-le-mono [of 2 p 2 q])
  by (auto)

lemma primmultgreater2: ⟦p ∈ prime; q ∈ prime; p ≠ q⟧ ⟹  2 < p*q
  apply (case-tac p=2)
  apply (simp)+
  apply (simp add: prime-def)
  apply (case-tac q=2)
  apply (simp add: prime-def)
  apply (erule primmultgreater)
  by (auto)

lemma cryptinverts: ⟦p ∈ prime; q ∈ prime; p ≠ q; n = p*q; m < n;
  e*d mod ((pred p)*(pred q)) = 1⟧ ⟹
  rsa-crypt (rsa-crypt (m,e,n), d , n) = m
  apply (insert cryptinverts-hilf4 [of p q m e*d])
  apply (insert cryptcorrect [of p*q rsa-crypt (m, e, p * q) d])
  apply (insert cryptcorrect [of p*q m e])
  apply (insert primmultgreater2 [of p q])
  apply (auto simp add: prime-def)
  by (auto simp add: remainderexp [of m^e p*q d] power-mult
  [THEN sym])

end
```

D Extensions to the Isabelle Word theory required for SHA1

```
theory WordOperations = Word + EfficientNat:

types
```

bv = *bit list*

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

consts
bvxor :: *bv* ⇒ *bv* ⇒ *bv*
bvand :: *bv* ⇒ *bv* ⇒ *bv*
bvor :: *bv* ⇒ *bv* ⇒ *bv*
bvrol :: *bv* ⇒ *nat* ⇒ *bv*
bvror :: *bv* ⇒ *nat* ⇒ *bv*
addmod32 :: *bv* ⇒ *bv* ⇒ *bv*
zerolist:: *nat* ⇒ *bv*
select :: *bv* ⇒ *nat* ⇒ *nat* ⇒ *bv*
hextobv :: *HEX* ⇒ *bv*
hexvtobv :: *HEX list* ⇒ *bv*
bv-prepend :: *nat* => *bit* => *bv* => *bv*
bvrolhelp :: *bv* × *nat* ⇒ *bv*
bvrorhelp :: *bv* × *nat* ⇒ *bv*
selecthelp1 :: *bv* × *nat* × *nat* ⇒ *bv*
selecthelp2 :: *bv* × *nat* ⇒ *bv*
reverse :: *bv* ⇒ *bv*
last :: *bv* ⇒ *bit*
dellast :: *bv* ⇒ *bv*

defs
bvxor:
bvxor a b == *bv-mapzip* (*op bitxor*) *a b*

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

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

bvrol:
bvrol x a == *bvrolhelp*(*x*,*a*)

bvror:
bvror x a == *bvrorhelp*(*x*,*a*)

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

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

primrec

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

defs
select:
select x i l == (*selecthelp1*(*x*,*i*,*l*))

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

recdef
bvrolhelp measure(λ(*a*,*x*).*x*)
bvrolhelp (*a*,*0*) = *a*
bvrolhelp ([],*x*) = []
bvrolhelp ((*x*#*r*),(*Suc n*)) = *bvrolhelp*((*r*@[*x*]),*n*)

recdef
bvrorhelp measure(λ(*a*,*x*).*x*)
bvrorhelp (*a*,*0*) = *a*
bvrorhelp ([],*x*) = []
bvrorhelp (*x*,(*Suc n*)) = *bvrorhelp*((*last x*)#(*dellast x*),*n*)

recdef
selecthelp1 measure(λ(*x*,*i*,*n*). *i*)
selecthelp1 ([],*i*,*n*) = (*if* (*i* <= *0*) *then* (*selecthelp2*([],*n*))
else (*selecthelp1*([],*i*−(*1*::*nat*),*n*−(*1*::*nat*))))
selecthelp1 (*x*#*l*,*i*,*n*) = (*if* (*i* <= *0*) *then* (*selecthelp2*(*x*#*l*,*n*))
else (*selecthelp1*(*l*,*i*−(*1*::*nat*),*n*−(*1*::*nat*))))

recdef
selecthelp2 measure(λ(*x*,*n*). *n*)

```
  selecthelp2 ([],n) = (if (n <= 0) then [Zero]
  else (Zero#selecthelp2([],n-(1::nat))))
  selecthelp2 (x#l,n) = (if (n <= 0) then [x]
  else (x#selecthelp2(l,(n-(1::nat)))))

primrec
  reverse [] = []
  reverse (x#r) = (reverse r)@[x]

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

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


lemma selectlenhelp: ALL l. length (selecthelp2(l,i)) = (i + 1)
proof
  show ⋀ l. length (selecthelp2 (l,i)) = i+1
  proof (induct i)
    fix l
    show length (selecthelp2 (l, 0)) = 0 + 1
    proof (cases l)
      case Nil
      hence selecthelp2(l, 0) = [Zero] by (simp)
      thus ?thesis by (simp)
    next
      case (Cons a list)
      hence selecthelp2(l, 0) = [a] by (simp)
      thus ?thesis by (simp)
    qed
  next
    fix l
    case (Suc x)
    show length (selecthelp2(l, (Suc x))) = (Suc x) + 1
    proof (cases l)
      case Nil
      hence (selecthelp2(l, (Suc x))) = Zero#selecthelp2(l, x)
        by (simp)
      thus length (selecthelp2(l, (Suc x))) = (Suc x) + 1 using Suc
        by (simp)
    next
      case (Cons a b)
      hence (selecthelp2(l, (Suc x))) = a#selecthelp2(b, x)
        by (simp)
      hence length (selecthelp2(l, (Suc x))) =
        1+(length (selecthelp2(b,x))) by (simp)
      thus length (selecthelp2(l, (Suc x))) = (Suc x) + 1 using Suc
```

```
        by (simp)
      qed
    qed
  qed

lemma selectlenhelp2:
  ⋀ i. ALL l j. EX k. selecthelp1(l,i,j) = selecthelp1(k,0,j−i)
proof (auto)
  fix i
  show ⋀ l j. ∃ k. selecthelp1 (l, i, j) = selecthelp1 (k, 0, j − i)
  proof (induct i)
    fix l and j
    have selecthelp1(l,0,j) = selecthelp1(l,0,j−(0::nat)) by (simp)
    thus EX k. selecthelp1 (l, 0, j) = selecthelp1 (k, 0, j − (0::nat))
      by (auto)
  next
    case (Suc x)
    have b: selecthelp1(l,(Suc x),j) = selecthelp1(tl l, x, j−(1::nat))
    proof (cases l)
      case Nil
      hence selecthelp1(l,(Suc x),j) = selecthelp1(l,x,j−(1::nat))
        by (simp)
      moreover have tl l = l using Nil by (simp)
      ultimately show ?thesis by (simp)
    next
      case (Cons head tail)
      hence selecthelp1(l,(Suc x),j) = selecthelp1(tail,x,j−(1::nat))
        by (simp)
      moreover have tail = tl l using Cons by (simp)
      ultimately show ?thesis by (simp)
    qed
    have ∃ k. selecthelp1 (l, x, j) = selecthelp1 (k, 0, j − (x::nat))
      using Suc by (simp)
    moreover have  EX k. selecthelp1(tl l,x,j−(1::nat)) =
      selecthelp1(k,0,j−(1::nat)−(x::nat))
      using Suc [of tl l j−(1::nat)] by auto
    ultimately have EX k. selecthelp1(l, Suc x, j) =
      selecthelp1(k,0,j−(1::nat) − (x::nat)) using b by (auto)
    thus EX k. selecthelp1 (l, Suc x, j) =
      selecthelp1 (k, 0 , j − (Suc x)) by (simp)
  qed
qed

lemma selectlenhelp3: ALL j. selecthelp1(l,0,j) = selecthelp2(l,j)
proof
  fix j
  show selecthelp1 (l, 0, j) = selecthelp2 (l, j)
  proof (cases l)
    case Nil
    assume l=[]
```

```
    thus selecthelp1 (l, 0, j) = selecthelp2 (l, j) by (simp)
  next
    case (Cons a b)
    thus selecthelp1(l,0,j) = selecthelp2(l,j) by (simp)
  qed
qed

lemma selectlenhelp4: length (selecthelp1(l,i,j)) = (j−i + 1)
proof −
  from selectlenhelp2 have
    EX k. selecthelp1(l,i, j) = selecthelp1(k,0,j−i) by (simp)
  hence EX k. length (selecthelp1(l, i, j)) =
    length (selecthelp1(k,0,j−i)) by (auto)
  hence c: EX k. length (selecthelp1(l, i, j)) =
    length (selecthelp2(k,j−i)) using selectlenhelp3 by (simp)
  from c obtain k where d: length (selecthelp1(l, i, j)) =
    length (selecthelp2(k,j−i)) by (auto)
  have 0<=j−i by (arith)
  hence length (selecthelp2(k,j−i)) = j−i+1 using selectlenhelp
    by (simp)
  thus length (selecthelp1(l,i,j)) = j−i+1 using d by (simp)
qed

lemma selectlen:length (select bv i j) = (j−i)+1
proof (simp add: select)
  from selectlenhelp4 show length (selecthelp1(bv,i,j)) = Suc (j−i)
    by (simp)
qed

lemma reverselen: length (reverse a) = length a
proof (induct a)
  show length (reverse []) = length [] by (simp)
next
  case (Cons a1 a2)
  have reverse (a1#a2) = reverse (a2)@[a1] by (simp)
  hence length (reverse (a1#a2)) = Suc (length (reverse (a2)))
    by (simp)
  thus length (reverse (a1#a2)) = length (a1#a2) using Cons
    by (simp)
qed

lemma addmod32len: ⋀ a b. length (addmod32 a b) = 32
proof (simp add: addmod32)
  fix a and b
  have length (select (reverse (nat-to-bv (bv-to-nat a +
    bv-to-nat b))) 0 31) = 32 using selectlen [of - 0 31] by (simp)
  thus length (reverse (select (reverse (nat-to-bv (bv-to-nat a +
    bv-to-nat b))) 0 31)) = 32 using reverselen by (simp)
qed
```

end

E Message Padding for SHA-1

theory *SHA1Padding* = *WordOperations*:

consts
sha1padd :: *bv* ⇒ *bv*
helppadd :: (*bv* × *bv* × *nat*) ⇒ *bv*
zerocount :: *nat* ⇒ *nat*

defs
sha1padd:
sha1padd x == *helppadd* (*x*,*nat-to-bv* (*length x*),(*length x*))

recdef *helppadd measure*(λ (*x*,*y*,*n*). *n*)
helppadd (*x*,*y*,*n*) = *x*@[*One*]@(*zerolist* (*zerocount n*))@
(*zerolist* (*64*−*length y*))@*y*

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

end

F Formal definition of the secure hash algorithm (SHA-1)

theory *SHA1* = *SHA1Padding*:

consts
sha1 :: *bv* ⇒ *bv*
sha1expand :: *bv* × *nat* ⇒ *bv*
sha1expandhelp :: *bv* × *nat* ⇒ *bv*
sha1block :: *bv* × *bv* × *bv* × *bv* × *bv* × *bv* × *bv* ⇒ *bv*
sha1compressstart :: *nat* ⇒ *bv* ⇒ *bv* ⇒ *bv* ⇒ *bv* ⇒ *bv* ⇒ *bv* ⇒ *bv*
sha1compress :: *nat* ⇒ *bv* ⇒ *bv* ⇒ *bv* ⇒ *bv* ⇒ *bv* ⇒ *bv* ⇒ *bv*
IV1 :: *bv*
IV2 :: *bv*
IV3 :: *bv*
IV4 :: *bv*
IV5 :: *bv*
K1 :: *bv*
K2 :: *bv*
K3 :: *bv*
K4 :: *bv*
kselect :: *nat* ⇒ *bv*
fif :: *bv* ⇒ *bv* ⇒ *bv* ⇒ *bv*

```
  fxor :: bv ⇒ bv ⇒ bv ⇒ bv
  fmaj :: bv ⇒ bv ⇒ bv ⇒ bv
  fselect :: nat ⇒ bv ⇒ bv ⇒ bv ⇒ bv
  getblock :: bv ⇒ bv
  delblock :: bv ⇒ bv
  delblockhelp :: bv × nat ⇒ bv

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

recdef
  sha1expand measure(λ(x,i). i)
  sha1expand (x,i) = (if (i < 16) then x else
  (let y = sha1expandhelp(x,i) in (sha1expand(x@y,i−(1::nat)))))

recdef
  sha1expandhelp measure(λ(x,i). i)
  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))

defs
  getblock:
  getblock x == select x 0 511

  delblock:
  delblock x == delblockhelp (x,512)

recdef delblockhelp measure (λ(x,n).n)
  delblockhelp ([],n) = []
  delblockhelp (x#r,n) = (if (n <= 0) then (x#r) else
  (delblockhelp (r,n−(1::nat))))

lemma sha1blockhilf: length (delblock (x#a)) < Suc (length a)
proof (simp add: delblock)
  have ⋀ n. length (delblockhelp (a,n)) <= length a
  proof −
    fix n
    show  length (delblockhelp (a,n)) <= length a
      by (induct n rule: delblockhelp.induct, auto)
  qed
  thus length (delblockhelp (a, 511)) < Suc (length a)
    using le-less-trans [of length (delblockhelp(a,511)) length a]
    by (simp)
qed
```

```
recdef sha1block measure(λ(b,x,A,B,C,D,E).length x)
  sha1block(b,[],A,B,C,D,E) = (let H = sha1compressstart 79 b A B C D E
                               in (let 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
                               in (let 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,EE)))
(hints recdef-simp:sha1blockhilf)

defs
  sha1compressstart:
  sha1compressstart r b A B C D E ==
  sha1compress r (sha1expand(b,79)) A B C D E

primrec
  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)))

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

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

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

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

  IV5:
```

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

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

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

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

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

  kselect:
  kselect r == (if (r < 20) then K1 else (if (r < 40) then K2
  else (if (r < 60) then K3 else K4)))

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

  fxor:
  fxor x y z == bvxor (bvxor x y) z

  fmaj:
  fmaj x y z == bvor (bvor (bvand x y) (bvand x z)) (bvand y z)

  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))))

lemma sha1blocklen: length (sha1block (b,x,A,B,C,D,E)) = 160
proof (induct b x A B C D E rule: sha1block.induct)
  show !!b A B C D E. length (sha1block (b, [], A, B, C, D, E)) = 160
    by (simp add: Let-def addmod32len)
  show !!b z aa A B C D E.
    ALL EE H DD CC BB AA.
    EE = addmod32 E (select H 128 159) &
    DD = addmod32 D (select H 96 127) &
    CC = addmod32 C (select H 64 95) &
    BB = addmod32 B (select H 32 63) &
    AA = addmod32 A (select H 0 31) &
    H = sha1compressstart 79 b A B C D E -->
    length (sha1block
    (getblock (z # aa), delblock (z # aa), AA, BB, CC, DD, EE)) = 160
    ==> length (sha1block (b, z # aa, A, B, C, D, E)) = 160
  by (simp add: Let-def)
qed
```

lemma *sha1len*: *length* (*sha1 m*) = *160*
proof (*simp add*: *sha1*)
  show *length* (*let y* = *sha1padd m*
    in sha1block (*getblock y*, *delblock y*, *IV1*, *IV2*, *IV3*, *IV4*, *IV5*)) =
    160 **by** (*simp add*: *sha1blocklen Let-def*)
qed

end

G Extensions to the Word theory required for PSS

theory *Wordarith* = *WordOperations* + *Primes*:

consts
  nat-to-bv-length :: *nat* ⇒ *nat* ⇒ *bv*
  roundup :: *nat* ⇒ *nat* ⇒ *nat*
  remzero :: *bv* ⇒ *bv*

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

  roundup:
  roundup x y == *if* (*x mod y* = *0*) *then* (*x div y*) *else* (*x div y*) + *1*

primrec
  remzero [] = []
  remzero (*a#b*) = (*if* (*a* = **1**) *then* (*a#b*) *else* (*remzero b*))

lemma *length-nat-to-bv-length* [*rule-format*]:
  nat-to-bv-length x y ≠ [] ⟶ *length* (*nat-to-bv-length x y*) = *y*
  by (*simp add*: *nat-to-bv-length*)

lemma *bv-to-nat-nat-to-bv-length* [*rule-format*]:
  nat-to-bv-length x y ≠ [] ⟶ *bv-to-nat* (*nat-to-bv-length x y*) = *x*
  by (*simp add*: *nat-to-bv-length*)

lemma *max-min*: *max* (*a*::*nat*) (*min b a*) = *a*
  apply (*case-tac a* < *b*)
  apply (*simp add*: *min-def*)
  by (*simp add*:*max-def*)

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

lemma *remzeroeq*: **shows** *bv-to-nat a* = *bv-to-nat* (*remzero a*)
proof (*induct a*)
  show *bv-to-nat* [] = *bv-to-nat* (*remzero* []) **by** *simp*

```
next
  case (Cons a1 a2)
  show bv-to-nat (a1#a2) = bv-to-nat (remzero (a1#a2))
  proof (cases a1)
    assume a: a1 = 0 hence bv-to-nat (a1#a2) = bv-to-nat a2
      by simp
    moreover have remzero (a1 # a2) = remzero a2 using a by simp
    ultimately show ?thesis using Cons by simp
  next
    assume a1 = 1 thus ?thesis by simp
  qed
qed

lemma len-nat-to-bv-pos:
  assumes x: 1 < a
  shows 0 < length (nat-to-bv a)
proof (auto)
  assume nat-to-bv a = []
  moreover have bv-to-nat [] = 0 by simp
  ultimately have bv-to-nat (nat-to-bv a) = 0 by simp
  moreover from x have bv-to-nat (nat-to-bv a) = a by simp
  ultimately have a=0 by simp
  thus False using x by simp
qed

lemma remzero-replicate: remzero ((replicate n 0)@l) = remzero l
by (induct n, auto)

lemma length-bvxor-bound: a ≤ length l ⟹ a ≤ length (bvxor l l2)
proof (induct a)
  show 0 ≤ length (bvxor l l2) by simp
next
  case (Suc a)
  assume a: Suc a ≤ length l
  hence b: a ≤ length (bvxor l l2) using Suc by simp
  thus Suc a ≤ length (bvxor l l2)
  proof (case-tac a = length (bvxor l l2))
    have length l ≤ max (length l) (length l2) by (simp add: max-def)
    hence Suc a ≤ max (length l) (length l2) using a by simp
    thus Suc a ≤ length (bvxor l l2) using bvxor by simp
  next
    assume a ≠ length (bvxor l l2)
    hence a < length (bvxor l l2) using b by simp
    thus ?thesis by simp
  qed
qed

lemma len-lower-bound:
  0 < n ⟹ 2^(length (nat-to-bv n) − Suc 0) ≤ n
proof (case-tac 1 < n)
```

```
assume 1 < n
thus 2 ^ (length (nat-to-bv n) − Suc 0) ≤ n
proof (simp add: nat-to-bv-def,induct n rule: nat-to-bv-helper.induct,
    auto)
  fix n
  assume a: Suc 0 < (n::nat) and b: ¬ Suc 0 < n div 2
  hence n = 2 ∨ n = 3
  proof (case-tac n ≤ 3)
    assume n ≤ 3 and Suc 0 < n
    thus n = 2 ∨ n = 3 by auto
  next
    assume ¬n ≤ 3 hence 3 < n by simp
    hence 1 < n div 2 by arith
    thus n = 2 ∨ n = 3 using b by simp
  qed
  thus 2 ^ (length (nat-to-bv-helper n []) − Suc 0) ≤ n
  proof (case-tac n = 2)
    assume a: n = 2 hence nat-to-bv-helper n [] = [1, 0]
    proof −
      have nat-to-bv-helper n [] = nat-to-bv n using b
        by (simp add: nat-to-bv-def)
      thus ?thesis using a by (simp add: nat-to-bv-non0)
    qed
    thus 2 ^ (length (nat-to-bv-helper n []) − Suc 0) ≤ n using a
      by simp
  next
    assume n = 2 ∨ n = 3 and n ≠ 2
    hence a: n=3 by simp
    hence nat-to-bv-helper n [] = [1, 1]
    proof −
      have nat-to-bv-helper n [] = nat-to-bv n using a
        by (simp add: nat-to-bv-def)
      thus ?thesis using a by (simp add: nat-to-bv-non0)
    qed
    thus 2^(length (nat-to-bv-helper n []) − Suc 0) ≤ n using a
      by simp
  qed
next
  fix n
  assume a: Suc 0<n and b: 2 ^ (length (nat-to-bv-helper
    (n div 2) []) − Suc 0) ≤ n div 2
  have (2::nat) ^ (length (nat-to-bv-helper n []) − Suc 0) =
    2^(length (nat-to-bv-helper (n div 2) []) + 1 − Suc 0)
  proof −
    have length (nat-to-bv n) = length (nat-to-bv (n div 2)) + 1
      using a by (simp add: nat-to-bv-non0)
    thus ?thesis by (simp add: nat-to-bv-def)
  qed
  moreover have (2::nat)^(length (nat-to-bv-helper (n div 2) []) +
    1 − Suc 0) = 2^(length (nat-to-bv-helper (n div 2) []) − Suc 0)*2
```

```
    proof auto
      have (2::nat)^(length (nat-to-bv-helper (n div 2) []) -Suc 0)*2 =
        2^(length (nat-to-bv-helper (n div 2) []) - Suc 0 + 1) by simp
      moreover have (2::nat)^(length (nat-to-bv-helper (n div 2) []) -
        Suc 0 + 1) = 2^(length (nat-to-bv-helper (n div 2) []))
      proof -
        have 0 < n div 2 using a by arith
        hence 0 < length (nat-to-bv (n div 2))
          by (simp add: nat-to-bv-non0)
        hence 0 < length (nat-to-bv-helper (n div 2) []) using a
          by (simp add: nat-to-bv-def)
        thus ?thesis by simp
      qed
      ultimately show
        (2::nat) ^ length (nat-to-bv-helper (n div 2) []) =
        2 ^ (length (nat-to-bv-helper (n div 2) []) - Suc 0) * 2
        by simp
    qed
    ultimately show 2 ^ (length (nat-to-bv-helper n []) - Suc 0) ≤ n
      using b by (simp add: nat-to-bv-def, arith)
  qed
next
  assume 0 < n and c: ¬ 1 < n
  thus 2 ^ (length (nat-to-bv n) - Suc 0) ≤ n
  proof (auto, case-tac n=1)
    assume a: n = 1 hence nat-to-bv n = [1]
      by (simp add: nat-to-bv-non0)
    thus 2^(length (nat-to-bv n) - Suc 0) ≤ n using a by simp
  next
    assume 0 < n and n ≠ 1 thus
      2^(length (nat-to-bv n) - Suc 0) ≤ n using c by simp
  qed
qed

lemma length-lower:
  assumes a: length a < length b and b: (hd b) ≠ 0
  shows bv-to-nat a < bv-to-nat b
proof -
  have ha: bv-to-nat a < 2^length a
    by (simp add: bv-to-nat-upper-range)
  have b ≠ [] using a by auto
  hence b = (hd b)#(tl b) by simp
  hence bv-to-nat b = bitval (hd b) * 2^(length (tl b)) +
    bv-to-nat (tl b) using bv-to-nat-helper [of hd b tl b] by simp
  moreover have bitval (hd b) = 1
  proof (cases hd b)
    assume hd b = 0
    thus  bitval (hd b) = 1 using b by simp
  next
    assume hd b = 1
```

```
    thus bitval (hd b) = 1 by simp
  qed
  ultimately have hb: 2^length (tl b) <= bv-to-nat b by simp
  have 2^(length a) ≤ (2::nat)^length (tl b) using a by (auto,arith)
  thus ?thesis using hb and ha by arith
qed

lemma nat-to-bv-non-empty:
  assumes a: 0 < n
  shows nat-to-bv n ≠ []
proof −
  from nat-to-bv-non0 [of n]
  have EX x. nat-to-bv n = x@[if n mod 2 = 0 then 0 else 1] using a
    by simp
  thus ?thesis by auto
qed

lemma hd-append: x ≠ [] ⟹ hd (x@y) = hd x
  by (induct x, auto)

lemma hd-one: 0 < n ⟹ hd (nat-to-bv-helper n []) = 1
proof (induct rule: nat-to-bv-helper.induct)
  fix n
  assume l: n ≠ 0 ⟶ 0 < n div 2 ⟶
    hd (nat-to-bv-helper (n div 2) []) = 1 and 0 < n
  thus  hd (nat-to-bv-helper n []) = 1
  proof (case-tac 1 < n)
    assume a: 1 < n hence n ≠ 0 by simp
    hence b: 0 < n div 2 ⟶ hd (nat-to-bv-helper (n div 2) []) = 1
      using l by simp
    from a have c: 0 < n div 2 by arith
    hence d: hd (nat-to-bv-helper (n div 2) []) = 1 using b by simp
    also from a have 0<n by simp
    hence hd (nat-to-bv-helper n []) = hd (nat-to-bv (n div 2) @
      [if n mod 2 = 0 then 0 else 1]) using nat-to-bv-def and
      nat-to-bv-non0 [of n] by auto
    hence hd (nat-to-bv-helper n []) = hd (nat-to-bv (n div 2))
      using nat-to-bv-non0 [of n div 2] and c and
        nat-to-bv-non-empty [of n div 2] and
        hd-append [of  nat-to-bv (n div 2)] by auto
    hence hd (nat-to-bv-helper n []) =
      hd (nat-to-bv-helper (n div 2) [])
      using nat-to-bv-def by simp
    thus hd (nat-to-bv-helper n []) = 1 using b and c by simp
  next
    assume ¬ 1 < n and 0 < n hence c: n = 1 by simp
    have (nat-to-bv-helper 1 []) = [1]
      by (simp add: nat-to-bv-helper.simps)
    thus hd (nat-to-bv-helper n []) = 1 using c by simp
  qed
```

qed

lemma *prime-hd-non-zero*:
  assumes *a*: *p* ∈ *prime* **and** *b*: *q* ∈ *prime*
  shows *hd* (*nat-to-bv* (*p*∗*q*)) ≠ **0**
proof −
  have *c*: ⋀ *p*. *p* ∈ *prime* ⟹ (*1*::*nat*) < *p*
  proof −
    fix *p*
    assume *d*: *p* ∈ *prime*
    thus *1* < *p* **by** (*simp add*: *prime-def*)
  qed
  have *1* < *p* **using** *c* **and** *a* **by** *simp*
  moreover have *1* < *q* **using** *c* **and** *b* **by** *simp*
  ultimately have *0* < *p*∗*q* **by** *simp*
  thus *?thesis* **using** *hd-one* [*of p*∗*q*] **and** *nat-to-bv-def* **by** *auto*
qed

lemma *primerew*: ⟦*m dvd p*; *m* ≠ *1*; *m* ≠ *p*⟧ ⟹ ¬ (*p* ∈ *prime*)
by (*auto simp add*: *prime-def*)

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

lemma *exp-prod1*: ⟦*1* < *b*; *2*ˆ*x* = *2*∗(*b*::*nat*)⟧ ⟹ *2 dvd b*
proof −
  assume *a*: *1* < *b* **and** *b*: *2*ˆ*x* = *2*∗(*b*::*nat*)
  have *s1*: *1* < *x*
  proof (*case-tac 1* < *x*)
    assume *1* < *x* **thus** *?thesis* **by** *simp*
  next
   assume *x*: ¬ *1* < *x* **hence** *2*ˆ*x* ≤ (*2*::*nat*) **using** *b*
   proof (*case-tac x* = *0*)
      assume *x* = *0* **thus** *2*ˆ*x* ≤ (*2*::*nat*) **by** *simp*
    next
      assume *x* ≠ *0* **hence** *x* = *1* **using** *x* **by** *simp*
      thus *2*ˆ*x* ≤ (*2*::*nat*) **by** *simp*
    qed
    hence *b* ≤ *1* **using** *b* **by** *simp*
    thus *?thesis* **using** *a* **by** *simp*
  qed
  have *s2*: *2*ˆ(*x* − (*1*::*nat*)) = *b*
  proof −
    from *s1* **have** *2*ˆ((*x* − *Suc 0*) + *1*) = *2*∗*b* **by** (*simp*)
    hence *2*∗*2*ˆ(*x* − *Suc 0*) = *2*∗*b* **by** *simp*
    thus *2*ˆ(*x* − (*1*::*nat*)) = *b* **by** *simp*
  qed
  from *s1* **and** *s2* **show** *?thesis* **using** *two-dvd-exp* [*of x* − (*1*::*nat*)]

```
  by simp
qed

lemma exp-prod2: ⟦1 < a; 2^x = a*2⟧ ⟹ (2::nat) dvd a
proof -
 assume 2^x = a*2
 hence 2^x = 2*a by simp
 moreover assume 1 < a
 ultimately show 2 dvd a using exp-prod1 by simp
qed

lemma odd-mul-odd: ⟦¬ (2::nat) dvd p; ¬ 2 dvd q⟧ ⟹ ¬ 2 dvd p*q
apply (simp add: dvd-eq-mod-eq-0)
by (simp add: mod-mult1-eq)

lemma prime-equal: ⟦p ∈ prime; q ∈ prime; 2^x = p*q⟧ ⟹ (p = q)
proof -
 assume a: p ∈ prime and b: q ∈ prime and c: 2^x = p*q
 from a have d: 1 < p by (simp add: prime-def)
 moreover from b have e: 1 < q by (simp add: prime-def)
 show p = q
 proof (case-tac p = 2)
  assume p: p = 2 hence 2 dvd q using c and
   exp-prod1 [of q x] and e by simp
  hence 2 = q using primerew [of 2 q] and b by auto
  thus ?thesis using p by simp
 next
  assume p: p ≠ 2 show p = q
  proof (case-tac q = 2)
   assume q: q = 2 hence 2 dvd p using c and
    exp-prod1 [of p x] and d by simp
   hence 2 = p using primerew [of 2 p] and a by auto
   thus ?thesis using p by simp
  next
   assume q: q ≠ 2 show p = q
   proof -
    from p have ¬ 2 dvd p using primerew and a by auto
    moreover from q have ¬ 2 dvd q using primerew and b
     by auto
    ultimately have ¬ 2 dvd p*q by (simp add: odd-mul-odd)
    moreover have (2::nat) dvd 2^x
    proof (case-tac x = 0)
     assume x = 0 hence (2::nat)^x = 1 by simp
     thus ?thesis using c and d and e by simp
    next
     assume x ≠ 0 hence 0 < x by simp
     thus ?thesis using two-dvd-exp by simp
    qed
    ultimately have 2^x ≠ p*q by auto
    thus ?thesis using c by simp
```

    qed
   qed
  qed
qed

lemma *nat-to-bv-length-bv-to-nat*[*rule-format*]:
  length xs = *n* ⟶ *xs* ≠ [] ⟶
  nat-to-bv-length (*bv-to-nat xs*) *n* = *xs*
  apply (*simp only*: *nat-to-bv-length*)
  apply (*auto*)
  by (*simp add*: *bv-extend-norm-unsigned*)

end

H EMSA-PSS encoding and decoding operation

theory *EMSAPSS* = *SHA1* + *Wordarith* + *Ring-and-Field*:

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 :: $bv \Rightarrow bv \Rightarrow bv \Rightarrow bv \Rightarrow nat \Rightarrow bool$
emsapss-decode-help6 :: $bv \Rightarrow bv \Rightarrow bv \Rightarrow nat \Rightarrow bool$
emsapss-decode-help7 :: $bv \Rightarrow bv \Rightarrow bv \Rightarrow nat \Rightarrow bool$
emsapss-decode-help8 :: $bv \Rightarrow bv \Rightarrow bv \Rightarrow bool$
emsapss-decode-help9 :: $bv \Rightarrow bv \Rightarrow bv \Rightarrow bool$
emsapss-decode-help10 :: $bv \Rightarrow bv \Rightarrow bool$
emsapss-decode-help11 :: $bv \Rightarrow bv \Rightarrow bool$

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* ∨ *2*ˆ*32*∗(*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 $(2\hat{}64 \leq$ *length M* $\vee$ $2\hat{}32 * 160 <$ *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 $(2\hat{}64 \leq$ *length M* $\vee$ $2\hat{}32{*}160{<}$*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* [*rule-format*]:
0 < emBits ⟶ 0 < (roundup emBits 160)
by (*simp add*: *roundup*, *safe*, *simp*)

lemma *roundup-ge-emBits* [*rule-format*]:
0 < emBits ⟶ 0 < x ⟶ emBits ≤ (roundup emBits x) ∗ x
apply (*simp add*: *roundup mult-commute*)
apply (*safe*)
apply (*simp*)
apply (*simp add*: *add-commute* [*of x x*∗(*emBits div x*)])
apply (*insert mod-div-equality2* [*of x emBits*])
apply (*subgoal-tac emBits mod x < x*)
apply (*arith*)
by (*simp only*: *mod-less-divisor*)

lemma *roundup-ge-0* [*rule-format*]:
0 < emBits ⟶ 0 < x ⟶ 0 ≤ (roundup emBits x) ∗ x − emBits
by (*simp add*: *roundup*)

lemma *roundup-le-7*:
0 < emBits ⟶ roundup emBits 8 ∗ 8 − emBits ≤ 7
apply (*simp add*: *roundup*)
apply (*insert div-mod-equality* [*of emBits 8 1*])
by (*arith*)

lemma *roundup-nat-ge-8-help* [*rule-format*]:
length (sha1 M) + sLen + 16 ≤ emBits ⟶
8 ≤ (roundup emBits 8) ∗ 8 − (length (sha1 M) + 8)
apply (*insert roundup-ge-emBits* [*of emBits 8*])
apply (*simp add*: *roundup sha1len sLen*)
apply (*safe*)
by (*simp*, *arith*)+

lemma *roundup-nat-ge-8* [*rule-format*]:
length (sha1 M) + sLen + 16 ≤ emBits ⟶
8 ≤ (roundup emBits 8) ∗ 8 − (length (sha1 M) + 8)
apply (*insert roundup-nat-ge-8-help* [*of M emBits*])
by (*arith*)

lemma *roundup-le-ub*: ⟦ *176 + sLen ≤ emBits*; *emBits ≤ 2ˆ32 ∗ 160*⟧ ⟹
(*roundup emBits 8*) ∗ *8 − 168 ≤ 2ˆ32 ∗ 160*
apply (*simp add*: *roundup*)
apply (*safe*)
apply (*simp*)
by (*arith*)+

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

lemma *modify-roundup-ge2*:
$[\![176 \leq roundup\ emBits\ 8 * 8]\!] \Longrightarrow 21 < roundup\ emBits\ 8$
by (*simp*)

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

lemma *roundup-help1-new*:
$[\![0 < l]\!] \Longrightarrow (roundup\ l\ 160 - 1) + 1 = (roundup\ l\ 160)$
apply (*drule roundup-positiv* [*of l*])
by (*arith*)

lemma *roundup-help2*:
$[\![176 + sLen \leq emBits]\!] \Longrightarrow roundup\ emBits\ 8 * 8 - emBits \leq$
$roundup\ emBits\ 8 * 8 - 160 - sLen - 16$
apply (*simp add*: *sLen*)
by (*arith*)

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

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

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

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

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

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

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

lemma *drop-bv-prepend-help2*:

⟦x < y⟧ ⟹ *drop x* (*bv-prepend y b* []) = *bv-prepend* (*y*−*x*) *b* []
apply (*insert bv-prepend-append2* [*of x y b* []])
by (*simp add*: *length-bv-prepend*)

lemma *drop-bv-prepend-help3*:
⟦x = y⟧ ⟹ *drop x* (*bv-prepend y b* []) = *bv-prepend* (*y*−*x*) *b* []
apply (*insert length-bv-prepend* [*of y b* []])
by (*simp add*: *bv-prepend*)

lemma *drop-bv-prepend-help4*:
⟦x ≤ y⟧ ⟹ *drop x* (*bv-prepend y b* []) = *bv-prepend* (*y*−*x*) *b* []
apply (*insert drop-bv-prepend-help2* [*of x y b*] *drop-bv-prepend-help3*
[*of x y b*])
by (*arith*)

lemma *bv-prepend-add*:
bv-prepend x b [] @ *bv-prepend y b* [] = *bv-prepend* (*x* + *y*) *b* []
apply (*induct-tac x*)
by (*simp add*: *bv-prepend*)+

lemma *bv-prepend-drop*: *x* ≤ *y* ⟶
bv-prepend x b (*drop x* (*bv-prepend y b* [])) = *bv-prepend y b* []
apply (*simp add*: *drop-bv-prepend-help4* [*of x y b*])
by (*simp add*: *bv-prepend-append* [*of x b* (*bv-prepend* (*y* − *x*) *b* [])]
bv-prepend-add)

lemma *bv-prepend-split*:
bv-prepend x b (*left* @ *right*) = *bv-prepend x b left* @ *right*
apply (*induct-tac x*)
by (*simp add*: *bv-prepend*)+

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

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

lemma *length-bvxor*[*rule-format*]:
length a = *length b* ⟶ *length* (*bvxor a b*) = *length a*
by (*simp add*: *bvxor*)

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

lemma *length-MGF1* [*rule-format*]:
l <= (*Suc n*) ∗ *160* ⟶ *length* (*MGF1 Z n l*) = *l*
apply (*simp add*: *MGF1 length-MGF2 sha1len*)

by (*arith*)

lemma *length-MGF*:
$[\![0 < l;\ l \leq\ 2\hat{}32 * length\ (sha1\ x)\ ]\!] \Longrightarrow length\ (MGF\ x\ l) = l$
apply (*simp add*: *MGF sha1len*)
apply (*insert roundup-help1-new* [*of l*])
apply (*rule length-MGF1*)
apply (*simp*)
apply (*insert roundup-ge-emBits* [*of l 160*])
by (*arith*)

lemma *solve-length-generate-DB*:
$[\![0 < emBits;\ length\ (sha1\ M) + sLen + 16 \leq emBits]\!] \Longrightarrow$
$length\ (generate\text{-}DB\ (generate\text{-}PS\ emBits\ (length\ (sha1\ x))\ )) =$
$(roundup\ emBits\ 8) * 8 - 168$
apply (*insert roundup-ge-emBits* [*of emBits 8*])
by (*simp add*: *length-generate-DB length-generate-PS sha1len*)

lemma *length-maskedDB-zero*:
$[\![roundup\ emBits\ 8 * 8 - emBits \leq length\ maskedDB]\!] \Longrightarrow$
$length\ (maskedDB\text{-}zero\ maskedDB\ emBits) = length\ maskedDB$
by (*simp add*: *maskedDB-zero length-bv-prepend*)

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

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

lemma *equal-zero*: $[\![176 + sLen \leq emBits;\ roundup\ emBits\ 8 * 8 -$
$emBits \leq roundup\ emBits\ 8 * 8 - (176 + sLen)]\!] \Longrightarrow 0 =$
$roundup\ emBits\ 8 * 8 - emBits - (roundup\ emBits\ 8 * 8 - (176 + sLen))$
by (*arith*)

lemma *get-salt*:
$[\![176 + sLen \leq emBits;\ roundup\ emBits\ 8 * 8 - emBits \leq 7]\!] \Longrightarrow$
$(generate\text{-}salt\ (maskedDB\text{-}zero\ (generate\text{-}DB\ (generate\text{-}PS$
$emBits\ 160))\ emBits)) = salt$
apply (*insert roundup-help2* [*of emBits*] *length-generate-PS* [*of emBits*]
equal-zero [*of emBits*])
apply (*simp add*: *generate-DB generate-PS maskedDB-zero*)
by (*simp add*: *bv-prepend-split bv-prepend-drop generate-salt*
show-rightmost-bits sLen)

lemma *generate-maskedDB-elim*: ⟦*roundup emBits 8 ∗ 8 − emBits ≤ length x*; (*roundup emBits 8*) ∗ *8* − (*length* (*sha1 M*)) − *8* = *length* (*maskedDB-zero x emBits*)⟧ ⟹
generate-maskedDB (*maskedDB-zero x emBits* @ *y* @ *z*) *emBits* (*length*(*sha1 M*)) = *maskedDB-zero x emBits*
apply (*simp add*: *maskedDB-zero*)
apply (*insert length-bv-prepend-drop* [*of* (*roundup emBits 8* ∗ *8* − *emBits*) *x*])
by (*simp add*: *generate-maskedDB*)

lemma *generate-H-elim*: ⟦*roundup emBits 8* ∗ *8* − *emBits* ≤ *length x*; *length* (*maskedDB-zero x emBits*) = (*roundup emBits 8*) ∗ *8* − *168*; *length y* = *160*⟧ ⟹
generate-H (*maskedDB-zero x emBits* @ *y* @ *z*) *emBits 160* = *y*
apply (*simp add*: *maskedDB-zero*)
apply (*insert length-bv-prepend-drop* [*of roundup emBits 8* ∗ *8* − *emBits x*])
by (*simp add*: *generate-H*)

lemma *length-bv-prepend-drop-special*:
⟦*roundup emBits 8* ∗ *8* − *emBits* ≤ *roundup emBits 8* ∗ *8* − (*176* + *sLen*); *length* (*generate-PS emBits 160*) = *roundup emBits 8* ∗ *8* − (*176* + *sLen*)⟧ ⟹ *length* (*bv-prepend* (*roundup emBits 8* ∗ *8* − *emBits*) **0** (*drop* (*roundup emBits 8* ∗ *8* − *emBits*) (*generate-PS emBits 160*))) = *length* (*generate-PS emBits 160*)
by (*simp add*: *length-bv-prepend-drop*)

lemma *x01-elim*:
⟦*176* + *sLen* ≤ *emBits*; *roundup emBits 8* ∗ *8* − *emBits* ≤ *7*⟧ ⟹
take 8 (*drop* (*roundup emBits 8* ∗ *8* − (*length* (*sha1 M*) + *sLen* + *16*)) (*maskedDB-zero* (*generate-DB* (*generate-PS emBits 160*)) *emBits*)) = [**0**, **0**, **0**, **0**, **0**, **0**, **0**, **1**]
apply (*insert roundup-help2* [*of emBits*] *length-generate-PS* [*of emBits*] *equal-zero* [*of emBits*])
by (*simp add*: *sha1len maskedDB-zero generate-DB generate-PS bv-prepend-split bv-prepend-drop*)

lemma *drop-bv-mapzip*:
assumes *n* <= *length x length x* = *length y*
shows *drop n* (*bv-mapzip f x y*) = *bv-mapzip f* (*drop n x*) (*drop n y*)
proof −
have !*x y*. *n* <= *length x* −−> *length x* = *length y* −−> *drop n* (*bv-mapzip f x y*) = *bv-mapzip f* (*drop n x*) (*drop n y*)
apply (*induct n*)
apply *simp*
apply *safe*
apply (*case-tac x*,*case-tac*[!] *y*,*auto*)
done
with *prems*
show *?thesis*

by *simp*
qed

lemma [*simp*]:
assumes *length a* = *length b*
shows *bvxor* (*bvxor a b*) *b* = *a*
proof −
have !*b*. *length a* = *length b* ⟶ *bvxor* (*bvxor a b*) *b* = *a*
apply (*induct a*)
apply (*auto simp add*: *bvxor*)
apply (*case-tac b*)
apply (*simp*)+
apply (*case-tac a1*)
apply (*case-tac a*)
apply (*safe*)
apply (*simp*)+
apply (*case-tac a*)
apply (*simp*)+
done
with *prems*
show *?thesis*
by *simp*
qed

lemma *bvxorxor-elim-help* [*rule-format*]:
assumes *x* ≤ *length a length a* = *length b*
shows *bv-prepend x* **0** (*drop x* (*bvxor* (*bv-prepend x* **0**
(*drop x* (*bvxor a b*))) *b*)) = *bv-prepend x* **0** (*drop x a*)
proof −
have (*drop x* (*bvxor* (*bv-prepend x* **0** (*drop x* (*bvxor a b*))) *b*))
= (*drop x a*)
apply (*unfold bvxor bv-prepend*)
apply (*cut-tac prems*)
apply (*insert length-replicate* [*of x* **0**])
apply (*insert length-drop* [*of x a*])
apply (*insert length-drop* [*of x b*])
apply (*insert length-bvxor* [*of drop x a drop x b*])
apply (*subgoal-tac length* (*replicate x* **0** @
drop x (*bv-mapzip op* $\oplus_b$ *a b*)) = *length b*)
apply (*subgoal-tac b* = (*take x b*)@(*drop x b*))
apply (*insert drop-bv-mapzip* [*of x* (*replicate x* **0** @
drop x (*bv-mapzip op* $\oplus_b$ *a b*)) *b op* $\oplus_b$])
apply (*simp*)
apply (*insert drop-bv-mapzip* [*of x a b op* $\oplus_b$])
apply (*simp*)
apply (*fold bvxor*)
apply (*simp-all*)
done
with *prems*
show *?thesis*

by (*simp*)
qed

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

lemma *verify*: ⟦(*emsapss-encode M emBits*) ≠ [];
EM=(*emsapss-encode M emBits*)⟧ ⟹ *emsapss-decode M EM emBits* = *True*
apply (*simp add*: *emsapss-decode emsapss-encode*)
apply (*safe*, *simp*+)
apply (*simp add*: *emsapss-decode-help1 emsapss-encode-help1*)
apply (*safe*, *simp*+)
apply (*simp add*: *emsapss-decode-help2 emsapss-encode-help2*)
apply (*safe*)
apply (*simp add*: *emsapss-encode-help3 emsapss-encode-help4*
emsapss-encode-help5 emsapss-encode-help6)
apply (*safe*)
apply (*simp add*: *emsapss-encode-help7 emsapss-encode-help8*
lastbits-BC [*THEN sym*])+
apply (*simp add*: *emsapss-decode-help3 emsapss-encode-help3*
emsapss-decode-help4 emsapss-encode-help4)
apply (*safe*)
apply (*insert roundup-le-7* [*of emBits*] *roundup-ge-0* [*of emBits 8*]
roundup-nat-ge-8 [*of M emBits*])
apply (*simp add*: *generate-maskedDB min-def emsapss-encode-help5*
emsapss-encode-help6)
apply (*safe*)
apply (*simp*)
apply (*simp add*: *emsapss-encode-help7*)
apply (*simp only*: *emsapss-encode-help8*)
apply (*simp only*: *maskedDB-zero*)
apply (*simp only*: *take-bv-prepend2*)
apply (*simp*)
apply (*simp add*: *emsapss-encode-help5 emsapss-encode-help6*)
apply (*safe*)
apply (*simp*)+
apply (*insert solve-length-generate-DB* [*of emBits M*
generate-M′ (*sha1 M*) *salt*] *roundup-le-ub* [*of emBits*])
apply (*insert length-MGF* [*of* (*roundup emBits 8*) ∗ *8* − *168*
(*sha1* (*generate-M′* (*sha1 M*) *salt*))])
apply (*insert modify-roundup-ge1* [*of emBits*] *modify-roundup-ge2*
[*of emBits*])
apply (*simp add*: *sha1len emsapss-encode-help7 emsapss-encode-help8*)
apply (*insert length-bvxor* [*of* (*generate-DB* (*generate-PS emBits 160*))
(*MGF* (*sha1* (*generate-M′* (*sha1 M*) *salt*))
((*roundup emBits 8*) ∗ *8* − *168*))])

apply (*insert generate-maskedDB-elim* [*of emBits*
(*bvxor* (*generate-DB* (*generate-PS emBits 160*)) (*MGF* (*sha1*
(*generate-M′* (*sha1 M*) *salt*)) ((*roundup emBits 8*) ∗ *8* − *168*)))
M sha1 (*generate-M′* (*sha1 M*) *salt*) *BC*])
apply (*insert length-maskedDB-zero* [*of emBits*
(*bvxor* (*generate-DB* (*generate-PS emBits 160*))(*MGF* (*sha1*
(*generate-M′* (*sha1 M*) *salt*)) ((*roundup emBits 8*) ∗ *8* − *168*)))])
apply (*insert generate-H-elim* [*of emBits* (*bvxor* (*generate-DB*
(*generate-PS emBits 160*))(*MGF* (*sha1* (*generate-M′* (*sha1 M*) *salt*))
(*roundup emBits 8* ∗ *8* − *168*)))
sha1 (*generate-M′* (*sha1 M*) *salt*) *BC*])
apply (*simp add*: *sha1len emsapss-decode-help5*)
apply (*simp only*: *emsapss-decode-help6 emsapss-decode-help7*)
apply (*insert bvxorxor-elim* [*of emBits*
(*generate-DB* (*generate-PS emBits 160*))
(*MGF* (*sha1* (*generate-M′* (*sha1 M*) *salt*))
((*roundup emBits 8*) ∗ *8* − *168*))])
apply (*fold maskedDB-zero*)
apply (*insert take-equal-bv-prepend* [*of emBits M*]
x01-elim [*of emBits M*] *get-salt* [*of emBits*])
by (*simp add*: *emsapss-decode-help8 emsapss-decode-help9*
emsapss-decode-help10 emsapss-decode-help11)

end

I RSA-PSS encoding and decoding operation

theory *RSAPSS* = *EMSAPSS* + *Cryptinverts*:

consts
rsapss-sign:: *bv* ⇒ *nat* ⇒ *nat* ⇒ *bv*
rsapss-sign-help1:: *nat* ⇒ *nat* ⇒ *nat* ⇒ *bv*
rsapss-verify:: *bv* ⇒ *bv* ⇒ *nat* ⇒ *nat* ⇒ *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*) ≠
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* [*rule-format*]:
emsapss-encode m x ≠ [] ⟶
length (*emsapss-encode m x*) = *roundup x 8* ∗ *8*
apply (*simp add*: *emsapss-encode*)
apply (*simp add*: *emsapss-encode-help1*)
apply (*simp add*: *emsapss-encode-help2*)
apply (*simp add*: *emsapss-encode-help3*)
apply (*simp add*: *emsapss-encode-help4*)
apply (*simp add*: *emsapss-encode-help5*)
apply (*simp add*: *emsapss-encode-help6*)
apply (*simp add*: *emsapss-encode-help7*)
apply (*simp add*: *emsapss-encode-help8*)
apply (*simp add*: *maskedDB-zero*)
apply (*simp add*: *length-generate-DB*)
apply (*simp add*: *sha1len*)
apply (*simp add*: *bvxor*)
apply (*simp add*: *length-generate-PS*)
apply (*simp add*: *length-bv-prepend*)
apply (*simp add*: *MGF*)
apply (*simp add*: *MGF1*)
apply (*simp add*: *length-MGF2*)
apply (*simp add*: *sha1len*)
apply (*simp add*: *length-generate-DB*)
apply (*simp add*: *length-generate-PS*)
apply (*simp add*: *BC*)
apply (*simp add*: *max-min*)
apply (*insert roundup-ge-emBits* [*of x 8*])
apply (*safe*)
by (*simp*)+

lemma *bv-to-nat-emsapss-encode-le*: *emsapss-encode m x* ≠ [] ⟹
bv-to-nat (*emsapss-encode m x*) < *2*ˆ(*roundup x 8* ∗ *8*)
apply (*insert length-emsapss-encode* [*of m x*])
apply (*insert bv-to-nat-upper-range* [*of emsapss-encode m x*])
by (*simp*)

lemma *length-helper1*: **shows** *length* (*bvxor* (*generate-DB*
(*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))
(*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*generate-DB* (*generate-PS* (*length*
(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))))@
sha1 (*generate-M′* (*sha1 m*) *salt*) @ *BC*)
= *length* (*bvxor* (*generate-DB*
(*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))

(*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*generate-DB* (*generate-PS* (*length*
(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))))) + *168*
proof −
have *a*: *length BC* = *8* **by** (*simp add*: *BC*)
have *b*: *length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)) = *160*
by (*simp add*: *sha1len*)
have *c*: ⋀ *a b c*. *length* (*a*@*b*@*c*) = *length a* + *length b* + *length c*
by *simp*
from *a* **and** *b* **show** *?thesis* **using** *c* **by** *simp*
qed

lemma *MGFLen-helper*: *MGF z l* ≠ [] ⟹ *l* ≤ *2ˆ32*∗(*length* (*sha1 z*))
proof (*case-tac 2ˆ32*∗*length* (*sha1 z*) < *l*)
assume *x*: *MGF z l* ≠ []
assume *a*: *2* ˆ *32* ∗ *length* (*sha1 z*) < *l*
hence *MGF z l* = []
proof (*case-tac l*=*0*)
assume *l*=*0*
thus *MGF z l* = [] **by** (*simp add*: *MGF*)
next
assume *l*~=*0*
hence (*l* = *0* ∨ *2ˆ32*∗*length*(*sha1 z*) < *l*) = *True* **using** *a* **by** *fast*
thus *MGF z l* = [] **apply** (*simp only*: *MGF*) **by** *simp*
qed
thus *?thesis* **using** *x* **by** *simp*
next
assume ¬ *2* ˆ *32* ∗ *length* (*sha1 z*) < *l*
thus *?thesis* **by** *simp*
qed

lemma *length-helper2*:
assumes *p*: *p* ∈ *prime* **and** *q*: *q* ∈ *prime* **and**
mgf: (*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*)) (*length*
(*generate-DB* (*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))) ≠ [] **and**
len: *length* (*sha1 M*) + *sLen* + *16* ≤
(*length* (*nat-to-bv* (*p* ∗ *q*))) − *Suc 0*
shows *length* ((*bvxor* (*generate-DB*
(*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))
(*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*generate-DB*
(*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))))) =
(*roundup* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *8*) ∗ *8* − *168*
proof −
have *a*: *length* (*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*generate-DB* (*generate-PS* (*length*

(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))) = (*length*
(*generate-DB* (*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))
proof −
have *0* < (*length* (*generate-DB*
(*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))
by (*simp add*: *generate-DB*)
moreover have (*length* (*generate-DB* (*generate-PS*
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))) ≤
2^32 ∗ *length* (*sha1* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))
using *mgf* **and** *MGFLen-helper* **by** *simp*
ultimately show *?thesis* **using** *length-MGF* **by** *simp*
qed
have *b*: *length* (*generate-DB* (*generate-PS*
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))) =
((*roundup* ((*length* (*nat-to-bv* (*p* ∗ *q*))) − *Suc 0*) *8*) ∗ *8* − *168*)
proof −
have *0* <= (*length* (*nat-to-bv* (*p* ∗ *q*))) − *Suc 0*
proof −
from *p* **have** *p2*: *1*<*p* **by** (*simp add*: *prime-def*)
moreover from *q* **have** *1*<*q* **by** (*simp add*: *prime-def*)
ultimately have *p*<*p*∗*q* **by** *simp*
hence *1*<*p*∗*q* **using** *p2* **by** *arith*
thus *?thesis* **using** *len-nat-to-bv-pos* **by** *simp*
qed
thus *?thesis* **using** *solve-length-generate-DB* **using** *len* **by** *simp*
qed
have *c*: *length* (*bvxor*
(*generate-DB* (*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))
(*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*generate-DB* (*generate-PS* (*length*
(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))))) =
roundup (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *8* ∗ *8* − *168*
using *a* **and** *b* **and** *length-bvxor* **by** *simp*
then show *?thesis* **by** *simp*
qed

lemma *emBits-roundup-cancel*: *emBits mod 8* ≠ *0* ⟹
(*roundup emBits 8*)∗*8* − *emBits* = *8* − (*emBits mod 8*)
apply (*auto simp add*: *roundup*)
by (*arith*)

lemma *emBits-roundup-cancel2*: *emBits mod 8* ≠ *0* ⟹
(*roundup emBits 8*) ∗ *8* − (*8* − (*emBits mod 8*)) = *emBits*

```
  apply (auto simp add: roundup)
  by (arith)

lemma length-bound: ⟦emBits mod 8 ≠ 0; 8 ≤ (length maskedDB)⟧ ⟹
  length (remzero ((maskedDB-zero maskedDB emBits)@a@b)) ≤
  length (maskedDB@a@b) − (8 − (emBits mod 8))
proof −
  assume a: emBits mod 8 ≠ 0
  assume len: 8 ≤ (length maskedDB)
  have b: ⋀ a. length (remzero a) ≤ length a
  proof −
    fix a
    show length (remzero a) ≤ length a
    proof (induct a)
      show (length (remzero [])) ≤ length [] by (simp)
    next
      case (Cons hd tl)
      show (length (remzero (hd#tl))) ≤ length (hd#tl)
      proof (cases hd)
        assume hd = 0
        hence remzero (hd#tl) = remzero tl by simp
        thus ?thesis using Cons by simp
      next
        assume hd = 1
        hence remzero (hd#tl) = hd#tl by simp
        thus ?thesis by simp
      qed
    qed
  qed
  from len
  show length (remzero (maskedDB-zero maskedDB emBits @ a @ b)) ≤
    length (maskedDB @ a @ b) − (8 − emBits mod 8)
  proof −
    have remzero(bv-prepend ((roundup emBits 8) ∗ 8 − emBits)
      0 (drop ((roundup emBits 8)∗8 − emBits) maskedDB)@a@b) =
      remzero ((drop ((roundup emBits 8)∗8 −emBits) maskedDB)@a@b)
      using remzero-replicate by (simp add: bv-prepend)
    moreover from emBits-roundup-cancel
    have roundup emBits 8 ∗ 8 − emBits = 8 − emBits mod 8
      using a by simp
    moreover have length ((drop (8−emBits mod 8) maskedDB)@a@b) =
      length (maskedDB@a@b) − (8−emBits mod 8)
    proof −
      show ?thesis using length-drop[of (8−emBits mod 8) maskedDB]
      proof (simp)
        have 0 <= emBits mod 8 by simp
        hence 8−(emBits mod 8) <= 8 by simp
        thus  length maskedDB − (8 − emBits mod 8) +
          (length a + length b) = length maskedDB +
          (length a + length b) − (8 − emBits mod 8) using len by arith
```

```
      qed
    qed
    ultimately show ?thesis using b
      [of (drop ((roundup emBits 8)∗8 − emBits) maskedDB)@a@b]
      by (simp add: maskedDB-zero)
  qed
qed

lemma length-bound2: 8 ≤ length ((bvxor (generate-DB (generate-PS
  (length (nat-to-bv (p ∗ q)) − Suc 0)
  (length (sha1 (generate-M′ (sha1 m) salt)))))
  (MGF (sha1 (generate-M′ (sha1 m) salt))
  (length (generate-DB (generate-PS (length (nat-to-bv (p ∗ q)) − Suc 0)
  (length (sha1 (generate-M′ (sha1 m) salt)))))))))
proof −
  have 8 ≤ length (generate-DB
    (generate-PS (length (nat-to-bv (p ∗ q)) − Suc 0)
    (length (sha1 (generate-M′ (sha1 m) salt)))))
    by (simp add: generate-DB)
  thus ?thesis using length-bvxor-bound by simp
qed

lemma length-helper:
  assumes p: p ∈ prime and q: q ∈ prime and
  x: (length (nat-to-bv (p ∗ q)) − Suc 0) mod 8 ≠ 0 and
  mgf: (MGF (sha1 (generate-M′ (sha1 m) salt)) (length
  (generate-DB (generate-PS (length (nat-to-bv (p ∗ q)) − Suc 0)
  (length (sha1 (generate-M′ (sha1 m) salt))))))) ≠ [] and
  len: length (sha1 M) + sLen + 16 ≤
  (length (nat-to-bv (p ∗ q))) − Suc 0
  shows length (remzero (maskedDB-zero (bvxor (generate-DB
  (generate-PS (length (nat-to-bv (p ∗ q)) − Suc 0)
  (length (sha1 (generate-M′ (sha1 m) salt)))))
  (MGF (sha1 (generate-M′ (sha1 m) salt))
  (length (generate-DB (generate-PS (length (nat-to-bv (p ∗ q)) − Suc 0)
  (length (sha1 (generate-M′ (sha1 m) salt))))))))
  (length (nat-to-bv (p ∗ q)) − Suc 0) @
  sha1 (generate-M′ (sha1 m) salt) @ BC))
  < length (nat-to-bv (p ∗ q))
proof −
  from mgf have round: 168 ≤
    roundup (length (nat-to-bv (p ∗ q)) − Suc 0) 8 ∗ 8
  proof (simp only: sha1len sLen)
    from len have 160 + sLen +16 ≤ length (nat-to-bv (p ∗ q)) − Suc 0
      by (simp add: sha1len)
    hence len1: 176 <= length (nat-to-bv (p ∗ q)) − Suc 0 by simp
    have length (nat-to-bv (p∗q)) − Suc 0 ≤
      (roundup (length (nat-to-bv (p ∗ q)) − Suc 0) 8) ∗ 8
      apply (simp only: roundup)
    proof (case-tac (length (nat-to-bv (p∗q)) − Suc 0) mod 8 = 0)
```

assume *len2*: (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *mod 8* = *0*
hence (*if* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *mod 8* = *0 then*
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *div 8 else*
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *div 8* + *1*) ∗ *8* =
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *div 8* ∗ *8* **by** *simp*
moreover have (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *div 8* ∗ *8* =
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) **using** *len2*
by (*auto simp add*: *div-mod-equality*
[*of length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0 8 0*])
ultimately show *length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0* ≤
(*if* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *mod 8* = *0 then*
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *div 8 else*
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *div 8* + *1*) ∗ *8* **by** *simp*
next
assume *len2*: (*length* (*nat-to-bv* (*p*∗*q*)) − *Suc 0*) *mod 8* ≠ *0*
hence (*if* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *mod 8* = *0 then*
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *div 8 else*
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *div 8* + *1*) ∗ *8* =
((*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *div 8* + *1*) ∗ *8* **by** *simp*
moreover have *length* (*nat-to-bv* (*p*∗*q*)) − *Suc 0* ≤
((*length* (*nat-to-bv* (*p*∗*q*)) − *Suc 0*) *div 8* + *1*)∗*8*
proof (*auto*)
have *length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0* =
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *div 8* ∗ *8* +
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *mod 8*
by (*simp add*: *div-mod-equality*
[*of length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0 8 0*])
moreover have
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *mod 8* < *8* **by** *simp*
ultimately show *length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0* ≤
8 + (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *div 8* ∗ *8* **by** *arith*
qed
ultimately show *length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0* ≤
(*if* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *mod 8* = *0 then*
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *div 8 else*
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *div 8* + *1*) ∗ *8* **by** *simp*
qed
thus *168* ≤ *roundup* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *8* ∗ *8*
using *len1* **by** *simp*
qed
from *x* **have** *a*: *length*
(*remzero* (*maskedDB-zero* (*bvxor* (*generate-DB* (*generate-PS*
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))
(*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*generate-DB* (*generate-PS* (*length*
(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))))
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) @
sha1 (*generate-M′* (*sha1 m*) *salt*) @ *BC*)) <= *length* ((*bvxor*

(*generate-DB* (*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))
(*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*generate-DB* (*generate-PS*
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))))) @
sha1 (*generate-M′* (*sha1 m*) *salt*) @ *BC*) − (*8* −
(*length* (*nat-to-bv* (*p*∗*q*)) − *Suc 0*) *mod 8*)
using *length-bound* **and** *length-bound2* **by** *simp*
have *b*: *length* (*bvxor* (*generate-DB* (*generate-PS*
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))
(*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*generate-DB* (*generate-PS* (*length*
(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))) @
sha1 (*generate-M′* (*sha1 m*) *salt*) @ *BC*) =
length (*bvxor* (*generate-DB* (*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) −
Suc 0) (*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))
(*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*)) (*length* (*generate-DB*
(*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))))) + *168*
using *length-helper1* **by** *simp*
have *c*: *length* (*bvxor* (*generate-DB* (*generate-PS* (*length*
(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) (*length* (*sha1* (*generate-M′*
(*sha1 m*) *salt*))))) (*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*generate-DB* (*generate-PS* (*length*
(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) (*length* (*sha1* (*generate-M′*
(*sha1 m*) *salt*)))))))) =
(*roundup* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *8*) ∗ *8* − *168*
using *p* **and** *q* **and** *length-helper2* **and** *mgf* **and** *len* **by** *simp*
from *a* **and** *b* **and** *c* **have** *length* (*remzero* (*maskedDB-zero* (*bvxor*
(*generate-DB* (*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))
(*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*generate-DB* (*generate-PS* (*length*
(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))))
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) @
sha1 (*generate-M′* (*sha1 m*) *salt*) @ *BC*)) ≤
roundup (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *8* ∗ *8* − *168* + *168* −
(*8* − (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *mod 8*) **by** *simp*
hence *length* (*remzero* (*maskedDB-zero* (*bvxor* (*generate-DB*
(*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))
(*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*generate-DB* (*generate-PS* (*length*
(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))))
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) @

sha1 (*generate-M′* (*sha1 m*) *salt*) @ *BC*)) ≤
roundup (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *8* ∗ *8* − (*8* −
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *mod 8*) **using** *round* **by** *simp*
moreover have *roundup* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *8* ∗ *8* −
(*8* − (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *mod 8*) =
(*length* (*nat-to-bv* (*p*∗*q*))−*Suc 0*)
using *x* **and** *emBits-roundup-cancel2* **by** *simp*
moreover have *0* < *length* (*nat-to-bv* (*p*∗*q*))
proof −
from *p* **have** *s*: *1* < *p* **by** (*simp add*: *prime-def*)
moreover from *q* **have** *1* < *q* **by** (*simp add*: *prime-def*)
ultimately have *p* < *p*∗*q* **by** *simp*
hence *1* < *p*∗*q* **using** *s* **by** *arith*
thus *?thesis* **using** *len-nat-to-bv-pos* **by** *simp*
qed
ultimately show *?thesis* **by** *arith*
qed

lemma *length-emsapss-smaller-pq*: ⟦*p* ∈ *prime*; *q* ∈ *prime*;
emsapss-encode m (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) ≠ [];
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *mod 8* ≠ *0*⟧ ⟹
length (*remzero* (*emsapss-encode m* (*length* (*nat-to-bv* (*p* ∗ *q*)) −
Suc 0))) < *length* (*nat-to-bv* (*p*∗*q*))
proof −
assume *a*: *emsapss-encode m* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) ≠
[] **and** *p*: *p* ∈ *prime* **and** *q*: *q* ∈ *prime* **and**
x: (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) *mod 8* ≠ *0*
have *b*: *emsapss-encode m* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) =
emsapss-encode-help1 (*sha1 m*)(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
proof (*simp only*: *emsapss-encode*)
from *a* **show** (*if* ((*2^64* ≤ *length m*) ∨
(*2^32* ∗ *160* < (*length* (*nat-to-bv* (*p*∗*q*)) − *Suc 0*))) *then* [] *else*
(*emsapss-encode-help1* (*sha1 m*) (*length* (*nat-to-bv* (*p*∗*q*)) −
Suc 0))) =
(*emsapss-encode-help1* (*sha1 m*) (*length* (*nat-to-bv* (*p*∗*q*)) − *Suc 0*))
by (*auto simp add*: *emsapss-encode*)
qed
have *c*: *length* (*remzero* (*emsapss-encode-help1* (*sha1 m*)
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*))) < *length* (*nat-to-bv* (*p*∗*q*))
proof (*simp only*: *emsapss-encode-help1*)
from *a* **and** *b* **have** *d*: (*if* ((*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) <
(*length* (*sha1 m*) + *sLen* + *16*)) *then* [] *else*
(*emsapss-encode-help2* (*generate-M′* (*sha1 m*) *salt*)
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*))) =
(*emsapss-encode-help2* ((*generate-M′* (*sha1 m*)) *salt*)
(*length* (*nat-to-bv* (*p*∗*q*)) − *Suc 0*))
by (*auto simp add*: *emsapss-encode emsapss-encode-help1*)
from *d* **have** *len*: *length* (*sha1 m*) + *sLen* + *16* ≤
(*length* (*nat-to-bv* (*p*∗*q*))) − *Suc 0*
proof (*case-tac length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0* <

```
    length (sha1 m) + sLen + 16)
  assume length (nat-to-bv (p ∗ q)) − Suc 0 <
    length (sha1 m) + sLen + 16
  hence len1: (if length (nat-to-bv (p ∗ q)) − Suc 0 <
    length (sha1 m) + sLen + 16 then [] else
    emsapss-encode-help2 (generate-M' (sha1 m) salt)
    (length (nat-to-bv (p ∗ q)) − Suc 0)) = [] by simp
  assume len2: (if length (nat-to-bv (p ∗ q)) − Suc 0 <
    length (sha1 m) + sLen + 16 then [] else
    emsapss-encode-help2 (generate-M' (sha1 m) salt)
    (length (nat-to-bv (p ∗ q)) − Suc 0)) =
    emsapss-encode-help2 (generate-M' (sha1 m) salt)
    (length (nat-to-bv (p ∗ q)) − Suc 0)
  from len1 and len2 and a and b
  show length (sha1 m) + sLen + 16 ≤
    length (nat-to-bv (p ∗ q)) − Suc 0
    by (auto simp add: emsapss-encode emsapss-encode-help1)
next
  assume ¬ length (nat-to-bv (p ∗ q)) − Suc 0 <
    length (sha1 m) + sLen + 16
  thus length (sha1 m) + sLen + 16 ≤
    length (nat-to-bv (p ∗ q)) − Suc 0 by simp
qed
have e: length (remzero (emsapss-encode-help2 (generate-M'
  (sha1 m) salt) (length (nat-to-bv (p ∗ q)) − Suc 0))) <
  length (nat-to-bv (p ∗ q))
proof (simp only: emsapss-encode-help2)
  show length (remzero
    (emsapss-encode-help3 (sha1 (generate-M' (sha1 m) salt))
    (length (nat-to-bv (p ∗ q)) − Suc 0)))
    < length (nat-to-bv (p ∗ q))
  proof (simp add: emsapss-encode-help3 emsapss-encode-help4
      emsapss-encode-help5)
    show length (remzero (emsapss-encode-help6 (generate-DB
      (generate-PS (length (nat-to-bv (p ∗ q)) − Suc 0)
      (length (sha1 (generate-M' (sha1 m) salt)))))
      (MGF (sha1 (generate-M' (sha1 m) salt)) (length
      (generate-DB (generate-PS (length (nat-to-bv (p ∗ q)) −
      Suc 0) (length (sha1 (generate-M' (sha1 m) salt)))))))
      (sha1 (generate-M' (sha1 m) salt))
      (length (nat-to-bv (p ∗ q)) − Suc 0))) <
      length (nat-to-bv (p ∗ q))
     proof (simp only: emsapss-encode-help6)
       from a and b and d
       have mgf: MGF (sha1 (generate-M' (sha1 m) salt))
         (length (generate-DB (generate-PS
         (length (nat-to-bv (p ∗ q)) − Suc 0)
         (length (sha1 (generate-M' (sha1 m) salt)))))) ≠ []
         by (auto simp add: emsapss-encode emsapss-encode-help1
           emsapss-encode-help2 emsapss-encode-help3
```

emsapss-encode-help4 emsapss-encode-help5
emsapss-encode-help6)
from *a* **and** *b* **and** *d*
have *f*: (*if MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*generate-DB* (*generate-PS* (*length*
(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))) = []
then [] *else* (*emsapss-encode-help7*
(*bvxor* (*generate-DB* (*generate-PS* (*length*
(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))
(*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*generate-DB* (*generate-PS* (*length*
(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))))
(*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*))) =
(*emsapss-encode-help7* (*bvxor* (*generate-DB* (*generate-PS*
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))
(*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*generate-DB* (*generate-PS* (*length*
(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))))
(*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*))
by (*auto simp add*: *emsapss-encode emsapss-encode-help1*
emsapss-encode-help2 emsapss-encode-help3
emsapss-encode-help4 emsapss-encode-help5
emsapss-encode-help6)
have *length* (*remzero* (*emsapss-encode-help7*
(*bvxor* (*generate-DB* (*generate-PS* (*length*
(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*) (*length* (*sha1*
(*generate-M′* (*sha1 m*) *salt*)))))
(*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*generate-DB* (*generate-PS* (*length*
(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))))
(*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*))) <
length (*nat-to-bv* (*p* ∗ *q*))
proof (*simp add*: *emsapss-encode-help7 emsapss-encode-help8*)
from *p* **and** *q* **and** *x* **show** *length*
(*remzero* (*maskedDB-zero* (*bvxor* (*generate-DB*
(*generate-PS* (*length* (*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*)))))
(*MGF* (*sha1* (*generate-M′* (*sha1 m*) *salt*))
(*length* (*generate-DB* (*generate-PS* (*length*
(*nat-to-bv* (*p* ∗ *q*)) − *Suc 0*)
(*length* (*sha1* (*generate-M′* (*sha1 m*) *salt*))))))))

```
          (length (nat-to-bv (p * q)) - Suc 0) @
          sha1 (generate-M' (sha1 m) salt) @ BC)) <
          length (nat-to-bv (p * q))
          using length-helper and len and mgf by simp
        qed
        then show length
         (remzero (if MGF (sha1 (generate-M' (sha1 m) salt))
         (length (generate-DB (generate-PS (length
         (nat-to-bv (p * q)) - Suc 0)
         (length (sha1 (generate-M' (sha1 m) salt)))))) = []
         then []
         else emsapss-encode-help7
         (bvxor (generate-DB (generate-PS (length
         (nat-to-bv (p * q)) - Suc 0)
         (length (sha1 (generate-M' (sha1 m) salt)))))
         (MGF (sha1 (generate-M' (sha1 m) salt))
         (length (generate-DB (generate-PS (length
         (nat-to-bv (p * q)) - Suc 0)
         (length (sha1 (generate-M' (sha1 m) salt))))))))
         (sha1 (generate-M' (sha1 m) salt))
         (length (nat-to-bv (p * q)) - Suc 0))) <
         length (nat-to-bv (p * q)) using f by simp
      qed
    qed
  qed
 from d and e show length (remzero (
  if length (nat-to-bv (p * q)) - Suc 0 <
  length (sha1 m) + sLen + 16 then []
  else emsapss-encode-help2 (generate-M' (sha1 m) salt)
  (length (nat-to-bv (p * q)) - Suc 0))) <
  length (nat-to-bv (p * q)) by simp
qed
from b and c show ?thesis by simp
qed

lemma bv-to-nat-emsapss-smaller-pq:
 assumes a: p ∈ prime and b: q ∈ prime and pneq: p ~= q and
 c: emsapss-encode m (length (nat-to-bv (p * q)) - Suc 0) ≠ []
 shows bv-to-nat (emsapss-encode m (length
 (nat-to-bv (p * q)) - Suc 0)) < p*q
proof -
 from a and b and c show ?thesis
 proof (case-tac 8 dvd ((length (nat-to-bv (p * q))) - Suc 0))
  assume d: 8 dvd ((length (nat-to-bv (p * q))) - Suc 0)
  hence 2 ^ (roundup (length (nat-to-bv (p * q)) - Suc 0) 8 * 8) <
   p*q
  proof -
   from d have e: roundup (length (nat-to-bv (p * q)) -
    Suc 0) 8 * 8 = length (nat-to-bv (p * q)) - Suc 0
    using rnddvd by simp
```

```
    have p*q = bv-to-nat (nat-to-bv (p*q)) by simp
    hence 2 ^ (length (nat-to-bv (p * q)) - Suc 0) < p*q
    proof -
      have 0<p*q
      proof -
        have 0<p using a by (simp add: prime-def, arith)
        moreover have 0<q using b by (simp add: prime-def, arith)
        ultimately show ?thesis by simp
      qed
      moreover have 2^(length (nat-to-bv (p*q)) - Suc 0) ~= p*q
      proof (case-tac 2^(length (nat-to-bv (p*q)) - Suc 0) = p*q)
        assume 2^(length (nat-to-bv (p*q)) - Suc 0) = p*q
        then have p=q using a and b and prime-equal by simp
        thus ?thesis using pneq by simp
      next
        assume 2^(length (nat-to-bv (p*q)) - Suc 0) ~= p*q
        thus ?thesis by simp
      qed
      ultimately show ?thesis using len-lower-bound [of p*q]
        by (simp)
    qed
    thus ?thesis using e by simp
  qed
  moreover from c have bv-to-nat (emsapss-encode m (length
    (nat-to-bv (p * q)) - Suc 0)) < 2 ^ (roundup (length
    (nat-to-bv (p * q)) - Suc 0)8 * 8 )
    using bv-to-nat-emsapss-encode-le
    [of m (length (nat-to-bv (p * q)) - Suc 0)] by auto
  ultimately show ?thesis by simp
next
  assume y: ~(8 dvd (length (nat-to-bv (p*q)) - Suc 0))
  thus ?thesis
  proof -
    from y have x: ~((length (nat-to-bv (p * q)) - Suc 0) mod 8 = 0)
      by (simp add: dvd-eq-mod-eq-0)
    from remzeroeq have d: bv-to-nat (emsapss-encode m (length
      (nat-to-bv (p * q)) - Suc 0)) = bv-to-nat (remzero
      (emsapss-encode m (length (nat-to-bv (p * q)) - Suc 0)))
      by simp
    from a and b and c and x and
      length-emsapss-smaller-pq [of p q m]
    have bv-to-nat (remzero (emsapss-encode m (length
      (nat-to-bv (p * q)) - Suc 0))) < bv-to-nat (nat-to-bv (p*q))
      using length-lower[of remzero (emsapss-encode m (length
        (nat-to-bv (p * q)) - Suc 0)) nat-to-bv (p * q)] and
        prime-hd-non-zero[of p q] by (auto)
    thus bv-to-nat (emsapss-encode m (length
      (nat-to-bv (p * q)) - Suc 0)) < p * q using d and bv-nat-bv
      by simp
  qed
```

qed
qed

lemma *rsa-pss-verify*: ⟦$p \in prime$; $q \in prime$; $p \neq q$; $n = p{*}q$;
$e{*}d$ *mod* $((pred\ p){*}(pred\ q)) = 1$; *rsapss-sign m e n* $\neq$ [];
$s =$ *rsapss-sign m e n*⟧ $\Longrightarrow$ *rsapss-verify m s d n* $=$ *True*
apply (*simp only*: *rsapss-sign rsapss-verify*)
apply (*simp only*: *rsapss-sign-help1*)
apply (*auto*)
apply (*simp add*: *length-nat-to-bv-length*)
apply (*simp add*: *Let-def*)
apply (*simp add*: *bv-to-nat-nat-to-bv-length*)
apply (*insert length-emsapss-encode*
[*of m* (*length* (*nat-to-bv* ($p * q$)) $-$ *Suc 0*)])
apply (*insert bv-to-nat-emsapss-smaller-pq* [*of p q m*])
apply (*simp add*: *cryptinverts*)
apply (*insert length-emsapss-encode*
[*of m* (*length* (*nat-to-bv* ($p * q$)) $-$ *Suc 0*)])
apply (*insert nat-to-bv-length-bv-to-nat*
[*of emsapss-encode m* (*length* (*nat-to-bv* ($p * q$)) $-$ *Suc 0*)
roundup (*length* (*nat-to-bv* ($p * q$)) $-$ *Suc 0*) $8 * 8$])
by (*simp add*: *verify*)

end