PROOF CHECKING THE RSA PUBLIC KEY ENCRYPTION ALGORITHM[1]

Robert S. Boyer and J Strother Moore

MR Classification Numbers: 03-04, 03B35, 10A25, 68C20, 68G15

> The development of mathematics toward greater precision has led, as is well known, to the formalization of large tracts of it, so that one can prove any theorem using nothing but a few mechanical rules. -- Godel [11]

> But formalized mathematics cannot in practice be written down in full, and therefore we must have confidence in what might be called the common sense of the mathematician ... We shall therefore very quickly abandon formalized mathematics ... -- Bourbaki [1]

1. Introduction

A formal mathematical proof is a finite sequence of formulas, each element of which is either an axiom or the result of applying one of a fixed set of mechanical rules to previous formulas in the sequence. It is thus possible to write a computer program to check mechanically whether a given sequence is a formal proof. However, formal proofs are rarely used. Instead, typical proofs in journal articles, textbooks, and day-to-day mathematical communication use informal notation and leave many of the steps to the reader's imagination. Nevertheless, by transcribing the sentences of the proof into a formal notation, it is sometimes possible to use today's automatic theorem-provers to fill in the gaps between published steps and thus mechanically check some published, informal proofs.

In this paper we illustrate this idea by mechanically checking the recently published proof of the invertibility of the public key encryption algorithm described by Rivest, Shamir, and Adleman [17]. We will briefly explain the idea of public key encryption to motivate the theorem proved.

In [17] a mathematical function, here called CRYPT, is defined. CRYPT(M,e,n) is the encryption of message M with key (e,n). The function has the following important properties:

1. It is easy to compute CRYPT(M,e,n).
2. CRYPT is "invertible", i.e., if M is encrypted with key (e,n) and then decrypted with key (d,n) the result is M. That is, CRYPT(CRYPT(M,e,n),d,n) = M, under suitable conditions on M, n, e and d.
3. Publicly revealing CRYPT and (e,n) does not reveal an easy way to compute (d,n). Public key encryption thus avoids the problem of distributing keys via secure means. Each user (e.g., a computer on a network) generates an encryption key and a corresponding decryption key, publicizes the encryption key to enable others to send private messages, and never distributes the decryption key.

The function defined in [17] is CRYPT(M,e,n) = M^e mod n; in addition, algorithms are given for constructing e, d, and n so that CRYPT has the three properties above. The first two properties are proved in [17]. The third property is not proved; instead the authors of [17] argue that "all the obvious approaches to breaking our system are at least as difficult as factoring n." Since there is no known

[1]The research reported here was supported by National Science Foundation Grant MCS-8202943 and Office of Naval Research Contract N00014-81-K-0634.

algorithm for efficiently factoring large composites, the security property of CRYPT is obtained by constructing n as the product of two very large (200 digit) primes.

In this paper we focus on mechanically checking the proofs of the first two properties. A precise statement of the "invertibility" property is: CRYPT(CRYPT(M,e,n),d,n) = M, if n is the product of two distinct primes p and q, M<n, and e and d are multiplicative inverses in the ring of integers modulo (p-1)*(q-1). Our mechanical proof of this theorem requires that we first prove many familiar theorems of number theory, including Fermat's theorem: $M^{p-1} \bmod p = 1$, when p is a prime and p$\not|$M.

2. A Sketch of the Theorem-Prover

The theorem-prover we use is the current version of the system described in [2]. The theorem-prover deals with a quantifier free first order logic providing equality, recursively defined functions, mathematical induction, and inductively constructed objects such as the natural numbers and finite sequences.

The theorem-prover is a large interactive computer program. The main inputs provided by the user are new recursive definitions and conjectures to prove.

Before a proposed definition is admitted as a new axiom, certain conditions are mechanically checked to assure that there exists one and only one function satisfying the definition. The most important condition is that there exist a measure of the arguments of the function that is decreasing in a well-founded sense in each recursive call in the definition. The mechanized definitional principle can guess simple measures and well-founded relations; more complicated ones can be supplied by the user. Once a candidate measure and relation are found, the mechanical theorem-prover is invoked to prove theorems sufficient to admit the proposed definition.

Given a conjecture to prove, the theorem-prover orchestrates the application of many proof techniques under heuristic control. The main proof techniques used are:

- simplification - The system applies axioms, definitions, and previously proved theorems as rewrite rules to simplify expressions. For example, if f is a defined function, it is sometimes useful to replace an instance of f(x) by the corresponding instance of the definition of f. To avoid looping the simplifier contains elaborate heuristics to control the use of recursive definitions. One of the main heuristics is to expand f(x) to introduce a recursive call provided the arguments to the call already occur in the conjecture. Axioms and previously proved theorems are also used as rewrite rules. For example, the theorem

  ```
  prime(p) -> [p|a*b  <-> (p|a v p|b)].
  ```

  is used to replace instances of p|a*b by (p|a v p|b) whenever the hypothesis prime(p) can be established by simplification. The simplifier also contains decision procedures for propositional calculus, equality, and those formulas of rational arithmetic that can be built up from variables, integers, +, -, =, <, and ~.

- elimination of undesirable function symbols - The system uses axioms and previously proved lemmas to eliminate certain function symbols from the conjecture being proved. For example, it is a theorem that for each natural number i and each positive integer j, there exist natural numbers r<j and q such that i=r+qj. By replacing i with r+qj, the system can transform the expression i mod j to simply r and i/j to simply q.

- strengthening the conjecture to be proved - It is frequently the case that to prove some theorem by induction it is necessary to prove a stronger theorem than that initially posed. Our system contains several heuristics for guessing stronger conjectures to try to prove. One heuristic involves "using" equality hypotheses by substituting one side for the other

elsewhere in the conjecture and then strengthening the conjecture by throwing away the equality hypothesis. Another heuristic replaces certain nonvariable expressions in the conjecture by new variables.

For example, consider proving $(n^i)^j = n^{i*j}$, by induction on j. The induction step is

$$(n^i)^j = n^{i*j} \;\rightarrow\; (n^i)^{j+1} = n^{i*(j+1)}.$$

The conclusion simplifies to $n^{i*}(n^i)^j = n^{i+i*j}$. The system then applies the first heuristic above, using and throwing away the equality hypothesis, to obtain the goal

$$n^{i*}n^{i*j} = n^{i+i*j}.$$

The second heuristic then produces the goal $n^{i*}n^k = n^{i+k}$ by replacing i*j with the new variable k. This final goal, a natural lemma about exponentiation, is then proved by a second appeal to induction.

- induction - When all else fails, it is useful to try mathematical induction. The selection of an "appropriate" induction is based on an analysis of the recursive functions mentioned in the conjecture. For example, since n! is recursively defined in terms of (n-1)!, when n is not 0, the presence of n! in a conjecture, p(n), suggests a simple induction on n. The base case is p(0). In the induction step, n is non-0, the inductive hypothesis is p(n-1), and the induction conclusion is p(n). Observe that the n! in the conclusion can now be expanded by the simplifier and will produce a term involving (n-1)!, about which we have a hypothesis. Similarly, since we define i mod j recursively in terms of (i-j) mod j, when $0<j\leq i$, the occurrence of i mod j in a conjecture suggests an inductive argument in which we suppose $0<j\leq i$ and take as an inductive hypothesis the conjecture with i replaced by i-j.

  Typically, a conjecture to be proved contains many different recursive functions and they each suggest different induction schemas. Our induction mechanism contains many heuristics for combining and choosing between the suggested inductions. That the inductions invented by the system are valid may be proved by considering the well-foundedness theorems proved when recursive functions are admitted.

Readers interested in more details of the theorem-prover should see [2], in which the system, as of May, 1978, is described at a level of detail sufficient to permit reproduction of our results. Several chapters of [2] are devoted to detailed annotated proofs by the system, including its proof of the uniqueness of prime factorizations. Improvements made to the system since the publication of [2] include the addition of the above mentioned decision procedures for equalities and simple arithmetic inequalities, the extension of the definitional principle to include reflexive functions as described in [16], and a metafunction facility permitting the incorporation of new simplifiers after they have been mechanically proved correct [3].

Finally, we have added a primitive "hint" facility so that the user can tell the theorem-prover how to prove a theorem when its heuristics lead it down blind alleys. There are two types of hints used in this paper. The first permits the user to say "use lemma x with instantiation y." The interpretation of this hint is to obtain the lemma named, instantiate its variables as directed by y, and add the resulting formula as a hypothesis to the conjecture being proved. The system then applies its usual heuristics. The second type of hint is "induct as suggested by the recursion in f" where f is a previously admitted recursive function.

The theorem-prover is automatic in the sense that once it begins a proof attempt, no user guidance is permitted. However, every time it accepts a definition or proves a theorem it stores the definition or theorem for future use. By presenting the theorem-prover with an appropriate sequence of lemmas to

prove, the user can "lead" it to proofs it would not otherwise discover. Thus, the distinction between a proof checker and an automatic theorem-prover blurs once the system remembers and uses previously proved facts. An automatic theorem-prover merely enables the user to leave out some of the routine proof steps. A sufficiently good automatic theorem-prover might enable the user to check an "informal" proof by presenting to the machine no more material than one would present to a human colleague.

When we began the encryption proofs we initialized the theorem-prover to the current version of the lemma library listed in Appendix A of [2]. The library contains several hundred previously proved theorems. Most of the theorems in this library were irrelevant to the encryption proofs (e.g., there are many theorems about list processing functions such as REVERSE, FLATTEN, and SORT). However, among the theorems in the library are many elementary facts about addition, multiplication, and integer division with remainder. The deepest number theory result in the library is the uniqueness of prime factorizations.

3. Correctness of CRYPT

To show that M^e mod n is easy to compute -- even when the numbers involved contain hundreds of digits -- Rivest, Shamir and Adleman exhibit an algorithm for computing it in order $\log_2(e)$ steps. Below we define CRYPT as a recursive version of their algorithm and prove that it computes the desired function. The notation e/2 below denotes integer division, i.e., the floor of the rational quotient.

The material contained in boxes in this paper represents material typed by the user and checked by the theorem-prover. The boxed material very closely resembles traditional "informal" proofs. To make this more obvious to readers unfamiliar with our formal notation, we have taken the liberty of transcribing the user type-in into conventional mathematical English. Use of the phrase "Hint:" in boxed material notes those occasions on which we gave the system explicit hints.

It may not be immediately obvious to the casual reader that each line follows from the previous ones. We have been careful to give the reader no more or less information than was given the machine and challenge the reader to do the machine’s job: verify each line of the boxed material.

Box 1

We define the encryption algorithm as the recursive function CRYPT:

```
Definition.
CRYPT(M,e,n)
  =
if e is not a natural number or is 0,
   then 1;
elseif e is even,
   then
   (CRYPT(M,e/2,n))^2 mod n;
else
   (M * (CRYPT(M,e/2,n)^2 mod n)) mod n.
```

Lemma. (x*(y mod n)) mod n = (x*y) mod n.

Corollary. (a*(b*(y mod n))) mod n = (a*(b*y)) mod n. (Hint: let x be a*b in the preceding lemma.)

Theorem. CRYPT(M,e,n) is equal to M^e mod n, provided n is not 1.

In section 6 we give the actual user type-in for the material in Box 1. In order to reinforce in the reader's mind the fact that the theorem-prover assents to these claims only after proving them we offer the following comments.

Before accepting the definition of CRYPT the theorem-prover guesses that e decreases in each recursive call and then proves it by showing that when e is a non-0 natural number, e/2 is strictly smaller than e.

CRYPT uses the "binary method" of computing M^e (see [15]), which is based on the observation:

$$M^e = \begin{cases} (M^{e/2})^2, & \text{if } e \text{ is even} \\ M*(M^{e/2})^2, & \text{if } e \text{ is odd} \end{cases}$$

However, by doing multiplications modulo n, CRYPT keeps the intermediate results manageably small and computes M^e mod n.

The first lemma above -- i.e., that (x*(y mod n)) mod n is (x*y) mod n -- establishes that the intermediate mods can be dropped. This lemma is obviously important in establishing that CRYPT computes M^e mod n. We brought this fact to the system's attention before even attempting to have the system prove properties of CRYPT.

How did the theorem-prover prove (x*(y mod n)) mod n = (x*y) mod n? It first tried simplification, but no known rewrites could be applied under our heuristics. The system then decided to eliminate (y mod n) by replacing y with r+nq, where r<n. To permit this, the system case split on whether y is a natural number and n is a positive integer. The "pathological" cases, where y was not numeric or n was nonpositive, yielded immediately to simplification. In the case where y was a natural number and n was positive, the system replaced y with r+nq, where r<n. Thus (y mod n) became r and the left hand side of the conjecture became x*r mod n. On the right, x*y became x*(r+nq). The simplifier then distributed the

multiplication over the addition (using a previously proved lemma in the library) and obtained (x*r + n*q*x) mod n, which was further rewritten to x*r mod n by the lemma that i+nj mod n is i mod n. The left and right hand sides were then identical. The machine spent about 23 seconds of cpu time on the proof in Interlisp-10 on a DEC 2060.

The corollary above, that (a*(b*(y mod n))) mod n is (a*(b*y)) mod n, follows trivially from the previous line, by letting x be (a*b) and applying the associativity of multiplication. Since this observation is uninteresting to a human proof checker, the need for it in our mechanical proof exposes a deficiency in our mechanical theorem-prover. Why is this line needed by the machine? The reason has to do with the order in which rewrite rules are applied. Consider the term ((a*b)*(y mod n)) mod n. The lemma just proved can be applied as a rewrite rule from left to right, to eliminate the intermediate mod and produce (a*b)*y mod n, to which we can then apply associativity to get a*(b*y) mod n. However, if we apply associativity first we obtain a*(b*(y mod n)) mod n, and we can no longer use the first lemma from left to right.[2] The second observation solves this problem.

We did not anticipate the machine's need for the corollary. Instead, immediately after proving the lemma we thought the machine could prove that CRYPT computes M^e mod n. We commanded it to do so and watched its proof attempt on the screen. (Imagine watching a colleague proving the theorem on the blackboard.) We saw the term (a*(b*(y mod n))) mod n arise and remain "unsimplified" even though we knew it was (a*(b*y)) mod n. At that point we interjected with the corollary.

We now consider the machine's acceptance of the final theorem in Box 1, the claim that CRYPT computes the desired function. The first time we submitted the claim we did not include the hypothesis that $N \neq 1$, because the hypothesis is not noted by Rivest, Shamir and Adleman, who imply that the algorithm always computes M^e mod n. However, the theorem-prover failed to prove the simpler conjecture and exhibited a formula showing that the encryption algorithm does not compute M^e mod n when e is 0 and n is 1. In practice, n is always larger than 1, so the additional hypothesis is no burden.

The theorem-prover proved the final claim by induction on e. The base case is that e is not a natural number or is 0. In the induction step, it supposes e is positive and assumes the conjecture for e/2. Observe that this induction is precisely the one suggested by the recursion in CRYPT. The proof required about 6 minutes of cpu time.

4. Fermat's Theorem

The proof of the invertibility of CRYPT in [17] assumes the reader is familiar with elementary number theory up through Fermat's theorem. While a production model proof checker for informal proofs would come factory equipped with a good number theory library, we had no such library when we began the encryption proofs. We therefore had the system prove the following theorems:

- Many elementary facts about remainder and exponentiation.
- Suppose p and q are distinct primes, a mod p = b mod p, and a mod q = b mod q. Then a mod p*q = b mod p*q.[3] Hence, under the additional hypothesis b<p*q, a mod p*q = b.

---

[2]This is the Knuth-Bendix problem in rewrite driven simplification. See [13] for an elegant solution to the problem in certain cases.

[3]Cf. Theorem 53 of Hardy and Wright's An Introduction to the Theory of Numbers.

- Suppose p is a prime and p does not divide M. Then M*x mod p = M*y mod p iff x mod p = y mod p.[4] Hence, by letting y be 1, if p is a prime, M*x mod p = M mod p iff either p|M or x mod p = 1.
- The Pigeon Hole Principle: If L is a sequence of length n, every element of L is a positive integer, no element occurs twice in L, and every element of L is less than or equal to n, then L is a permutation of the sequence [n n-1 ... 2 1].
- The following straightforward observations about permutations and the concept of the product of the elements in a sequence:
    - If L1 is a permutation of L2 then the product of the elements in L1 is equal to that of the elements in L2.
    - The product of the elements in [n n-1 ... 2 1] is n!.
    - Hence, if L is a permutation of [n n-1 ... 2 1] then the product of the elements in L is n!.
    - If p is a prime and n<p, then p does not divide n!.

We then had the theorem-prover check the proof of Fermat's theorem in [14].

Box 2

Definition. We define S(n,M,p) to be the sequence:

[n*M mod p, (n-1)*M mod p, ..., 1*M mod p].

Lemma 1. The product of the elements in S(n,M,p) mod p is equal to $n!*M^n$ mod p.

Lemma. If p is a prime that does not divide M and i<j<p, then j*M mod p is not a member of S(i,M,p). (Hint: induct on i.)

Lemma. If p is a prime that does not divide M and n<p, then no element of S(n,M,p) occurs twice.

Lemma. If p is a prime that does not divide M and n<p, then each element of S(n,M,p) is a positive integer.

Lemma. If p>0, then each element of S(n,M,p) is less than or equal to p-1.

Lemma. S(n,M,p) has n elements.

Fermat's Theorem. If p is a prime that does not divide M then M^{p-1} mod p = 1.

(Hint: From Lemma 1 we have that the product of the elements in S(p-1,M,p) mod p is $(p-1)!*M^{p-1}$ mod p. But from the Pigeon Hole Principle we have that S(p-1,M,p) is a permutation of [p-1, ... 2, 1].)

To prove Lemma 1 the system inducts on n and uses the previously proved lemma that intermediate mods can be dropped.

To get the system to prove the next lemma, that j*M mod p is not a member S(i,M,p), we had to instruct it to induct on i. (If left to its own devices, it chooses here to induct on i and j simultaneously.) The proof is as follows. The base case, when i=0, is easy because S(0,M,p) is empty. In the induction step we assume that j*M mod p is not a member of S(i-1,M,p) and must prove that it is not a member of S(i,M,p). But by definition S(i,M,p) is the sequence consisting of i*M mod p followed by the sequence S(i-1,M,p).

[4]Cf. Theorem 55 of Hardy and Wright.

The induction hypothesis establishes that j*M mod p is not an element of the latter. It suffices to prove that j*M mod p ≠ i*M mod p. Suppose the contrary. Then j mod p = i mod p, since p∤M. Thus j=i, contradicting i<j.

The remaining lemmas above are proved by similar inductions of the system's own invention.

The system's proof of the main theorem is then as follows. The hints lead it to conclude that $(p-1)!*M^{p-1}$ mod p = (p-1)! mod p. Since p does not divide (p-1)!, we can cancel (p-1)! from both sides and get M^{p-1} mod p = 1.

5. Invertibility of CRYPT

We now prove that CRYPT(CRYPT(M,e,n),d,n) = M, if n is the product of two distinct primes p and q, M<n, and e and d are multiplicative inverses in the ring of integers modulo (p-1)*(q-1).

Box 3

Lemma 2. For all primes p, $(M*M^{k*(p-1)})$ mod p = M mod p.

Corollary. If p and q are prime, then

$$(M*M^{k*(p-1)*(q-1)}) \bmod p = M \bmod p$$

and

$$(M*M^{k*(p-1)*(q-1)}) \bmod q = M \bmod q.$$

(Hint: take two instantiations of (2).)

Lemma 3. If p and q are distinct primes, M is a natural number less than p*q, and x mod (p-1)*(q-1) is 1, then M^x mod p*q = M.

RSA Theorem. If p and q are distinct primes, n is p*q, M is a natural number less than n and e*d mod (p-1)*(q-1) is 1, CRYPT(CRYPT(M,e,n),d,n) = M.

The proof of Lemma 2 can be seen by rearranging the exponents and mods so that $(M*M^{k*(p-1)})$ mod p becomes $(M*(M^{p-1} \bmod p)^k \bmod p) \bmod p$. Fermat's Theorem can then be used to replace M^{p-1} mod p by 1. The corollary is obvious.

To prove Lemma 3 the system first observes that, for some k, x is k*(p-1)*(q-1) + 1. Thus, M^x mod p*q is $(M*M^{k*(p-1)*(q-1)})$ mod p*q. Now recall the previously mentioned result that if p and q are distinct primes and a mod p = b mod p and a mod q = b mod q then a mod p*q = b mod p*q. Letting a be $M*M^{k*(p-1)*(q-1)}$ and b be M and appealing to the corollary above, the system concludes that M^x mod p*q is M mod p*q, which, in turn, is M since M<p*q.

Finally, to prove the RSA Theorem itself, the system appeals to the correctness of CRYPT and the hypothesis that n=p*q to reduce the conclusion to $(M^e \bmod p*q)^d \bmod p*q = M$. It then eliminates the intermediate mod, collects the exponents e and d and appeals to Lemma 3.

6. Sample Input to the Theorem-Prover

To illustrate the sense in which the boxed material is an English transcription of the user supplied type-in to our theorem-prover, we give below the type-in for the material in Box 1. We use the prefix syntax of Church's lambda calculus and McCarthy's LISP. It would be straightforward to arrange for the system to read and print according to a more elaborate grammar, but we prefer the simplicity of prefix notation.

```
Definition.
(CRYPT M E N)
  =
(IF (ZEROP E)
    1
    (IF (EVEN E)
        (REMAINDER (SQUARE (CRYPT M (QUOTIENT E 2) N))
                   N)
        (REMAINDER
         (TIMES M
          (REMAINDER (SQUARE (CRYPT M (QUOTIENT E 2) N))
                     N))
         N)))

Theorem.  TIMES.MOD.1 (rewrite):
(EQUAL (REMAINDER (TIMES X (REMAINDER Y N)) N)
       (REMAINDER (TIMES X Y) N))

Theorem.  TIMES.MOD.2 (rewrite):
(EQUAL (REMAINDER (TIMES A (TIMES B (REMAINDER Y N)))
                  N)
       (REMAINDER (TIMES A B Y) N))
Hint:  Use TIMES.MOD.1 with X replaced by (TIMES A B).

Theorem.  CRYPT.CORRECT (rewrite):
(IMPLIES (NOT (EQUAL N 1))
         (EQUAL (CRYPT M E N) (REMAINDER (EXP M E) N)))
```

Readers interested in the complete list of definitions and theorems typed by the user, should see section 8 of [6].

7. Conclusion

We have shown how an existing mechanical theorem-prover was used to check a recently published proof. Among the other mathematically interesting proofs performed by our theorem-prover are:

- Wilson's Theorem: if p is a prime then (p-1)! mod p = p-1; [18]
- the termination over the integers of the Takeuchi function [16]:

```
tak(x,y,z) = if x≤y
                then y
                else tak(tak(x-1,y,z),
                         tak(y-1,z,x),
                         tak(z-1,x,y))
```

- the soundness and completeness of a decision procedure for propositional calculus [2];
- the existence of nonprimitive recursive functions;

- the Turing completeness of the Pure LISP programming language [8]; and
- the recursive unsolvability of the halting problem for Pure LISP [7].

We take these examples as evidence that proof checking mathematics is not only a theoretical but also a practical possibility. We doubt that the mechanical theorem-provers of today could be easily used to check theorems at the frontiers of mathematics. The less ambitious motivation behind much automatic theorem-proving research -- certainly ours -- is to mechanize the often mundane and tedious proofs arising in connection with computer programs. For example, our theorem-prover has been used to prove thousands of theorems related to the correctness of various programs [4, 5], communications protocols [9], and computer security [10]. Because of the high cost of bugs in software, the increasing impact of software due to cheap microprocessors, and the relatively shallow nature of most program correctness proofs, we expect to see, within the decade, commercial use of mechanical theorem-provers and formal logic in software development. The construction of an automatic theorem-prover that can advance the frontiers of mathematics, however, must still await another Godel or Herbrand.

8. The Formal Details

We list here all of the user commands typed to lead the theorem-prover from its initial library to the correctness and invertibility of the RSA algorithm.

8.1. Correctness of CRYPT

This section contains the formalization of the proof in Box 1.

```
1.   Definition.
     (CRYPT M E N)
        =
     (IF
      (ZEROP E)
      1
      (IF
       (EVEN E)
       (REMAINDER (SQUARE (CRYPT M (QUOTIENT E 2) N))
                  N)
       (REMAINDER
        (TIMES M
               (REMAINDER (SQUARE (CRYPT M (QUOTIENT E 2) N))
                          N))
        N)))

2.   Theorem.  TIMES.MOD.1 (rewrite):
     (EQUAL (REMAINDER (TIMES X (REMAINDER Y N)) N)
            (REMAINDER (TIMES X Y) N))

3.   Theorem.  TIMES.MOD.2 (rewrite):
     (EQUAL (REMAINDER (TIMES A (TIMES B (REMAINDER Y N)))
                       N)
            (REMAINDER (TIMES A B Y) N))
     Hint:  Consider:
            TIMES.MOD.1 with {X/(TIMES A B)}

4.   Theorem.  CRYPT.CORRECT (rewrite):
     (IMPLIES (NOT (EQUAL N 1))
              (EQUAL (CRYPT M E N)
                     (REMAINDER (EXP M E) N)))
```

8.2. Miscellaneous Theorems

We now lay some ground work used throughout the rest of the proofs. These lemmas represent the "elementary properties of remainder and exponentiation" mentioned in the informal proofs. The first important result is event 7, which states that $(a \bmod n)^i \bmod n$ is $(a^i) \bmod n$.

```
5.   Theorem.  TIMES.MOD.3 (rewrite):
     (EQUAL (REMAINDER (TIMES (REMAINDER A N) B) N)
            (REMAINDER (TIMES A B) N))

6.   Theorem.  REMAINDER.EXP/LEMMA (rewrite):
     (IMPLIES (EQUAL (REMAINDER Y A)
                     (REMAINDER Z A))
              (EQUAL (EQUAL (REMAINDER (TIMES X Y) A)
                            (REMAINDER (TIMES X Z) A))
                     T))
```

```
7.   Theorem.  REMAINDER.EXP (rewrite):
     (EQUAL (REMAINDER (EXP (REMAINDER A N) I) N)
            (REMAINDER (EXP A I) N))
```

We now proceed to teach the system several commonly used tricks. The first, event 8, shows the system that m^{i*j} mod n is 1 if m^j mod n is 1. To prove this obvious fact one must use the rewrite rules EXP.EXP and REMAINDER.EXP "against the grain" of the directed equality.

```
8.   Theorem.  EXP.MOD.IS.1 (rewrite):
     (IMPLIES (EQUAL (REMAINDER (EXP M J) P) 1)
              (EQUAL (REMAINDER (EXP M (TIMES I J)) P)
                     1))
     Hints:  Consider:
             EXP.EXP with {I/M, J/J, K/I}
             REMAINDER.EXP with {A/(EXP M J), N/P}
```

We next teach the system the trick of establishing (a mod p)=(b mod p) by considering whether p divides |a-b|, and vice versa. We define PDIFFERENCE, the absolute value of the integer difference of two naturals, in terms of our function DIFFERENCE, which returns 0 when the subtrahend is larger than the minuend. We then prove the necessary theorems about PDIFFERENCE and, then at event 13, we tell the system henceforth not to expand the definition of PDIFFERENCE.

```
9.   Definition.
     (PDIFFERENCE A B)
        =
     (IF (LESSP A B)
         (DIFFERENCE B A)
         (DIFFERENCE A B))

10.  Theorem.  TIMES.DISTRIBUTES.OVER.PDIFFERENCE (rewrite):
     (EQUAL (TIMES M (PDIFFERENCE A B))
            (PDIFFERENCE (TIMES M A) (TIMES M B)))

11.  Theorem.  EQUAL.MODS.TRICK.1 (rewrite):
     (IMPLIES (EQUAL (REMAINDER (PDIFFERENCE A B) P)
                     0)
              (EQUAL (EQUAL (REMAINDER A P)
                            (REMAINDER B P))
                     T))

12.  Theorem.  EQUAL.MODS.TRICK.2 (rewrite):
     (IMPLIES (EQUAL (REMAINDER A P)
                     (REMAINDER B P))
              (EQUAL (REMAINDER (PDIFFERENCE A B) P)
                     0))
     Hint:  Disable DIFFERENCE.ELIM

13.  Disable PDIFFERENCE
```

We conclude this subsection by showing the system one last trick: to prove that (a mod p)=(b mod p), when p is prime, find an m such that p does not divide m and (m*a mod p)=(m*b mod p).

```
14.  Theorem.  PRIME.KEY.TRICK (rewrite):
     (IMPLIES (AND (EQUAL (REMAINDER (TIMES M A) P)
                          (REMAINDER (TIMES M B) P))
                   (NOT (EQUAL (REMAINDER M P) 0))
                   (PRIME P))
              (EQUAL (EQUAL (REMAINDER A P)
                            (REMAINDER B P))
                     T))
     Hints:  Consider:
             PRIME.KEY.REWRITE with {A/M, B/(PDIFFERENCE A B)}
             EQUAL.MODS.TRICK.2 with {A/(TIMES M A),
                                      B/(TIMES M B)}
```

8.3. Theorems 53 and 55

We now prove versions of Theorems 53 and 55 from [12] and observe trivial corollaries of each. Event 15 is used in the proof of event 16, which is used in the proof of Theorem 53.

```
15.  Theorem.  PRODUCT.DIVIDES/LEMMA (rewrite):
     (IMPLIES (EQUAL (REMAINDER X Z) 0)
              (EQUAL (REMAINDER (TIMES Y X) (TIMES Y Z))
                     0))

16.  Theorem.  PRODUCT.DIVIDES (rewrite):
     (IMPLIES (AND (EQUAL (REMAINDER X P) 0)
                   (EQUAL (REMAINDER X Q) 0)
                   (PRIME P)
                   (PRIME Q)
                   (NOT (EQUAL P Q)))
              (EQUAL (REMAINDER X (TIMES P Q)) 0))

17.  Theorem.  THM.53.SPECIALIZED.TO.PRIMES:
     (IMPLIES (AND (PRIME P)
                   (PRIME Q)
                   (NOT (EQUAL P Q))
                   (EQUAL (REMAINDER A P)
                          (REMAINDER B P))
                   (EQUAL (REMAINDER A Q)
                          (REMAINDER B Q)))
              (EQUAL (REMAINDER A (TIMES P Q))
                     (REMAINDER B (TIMES P Q))))

18.  Theorem.  COROLLARY.53 (rewrite):
     (IMPLIES (AND (PRIME P)
                   (PRIME Q)
                   (NOT (EQUAL P Q))
                   (EQUAL (REMAINDER A P)
                          (REMAINDER B P))
                   (EQUAL (REMAINDER A Q)
                          (REMAINDER B Q))
                   (NUMBERP B)
                   (LESSP B (TIMES P Q)))
              (EQUAL (EQUAL (REMAINDER A (TIMES P Q)) B)
                     T))
     Hint:  Consider:
            THM.53.SPECIALIZED.TO.PRIMES
```

```
19.  Theorem.  THM.55.SPECIALIZED.TO.PRIMES (rewrite):
     (IMPLIES (AND (PRIME P)
                   (NOT (EQUAL (REMAINDER M P) 0)))
              (EQUAL (EQUAL (REMAINDER (TIMES M X) P)
                            (REMAINDER (TIMES M Y) P))
                     (EQUAL (REMAINDER X P)
                            (REMAINDER Y P))))

20.  Theorem.  COROLLARY.55 (rewrite):
     (IMPLIES (PRIME P)
              (EQUAL (EQUAL (REMAINDER (TIMES M X) P)
                            (REMAINDER M P))
                     (OR (EQUAL (REMAINDER M P) 0)
                         (EQUAL (REMAINDER X P) 1))))
     Hint:  Consider:
            THM.55.SPECIALIZED.TO.PRIMES with {Y/1}
```

8.4. The Pigeon Hole Principle

We are now on our way to Fermat's theorem and must state formally and prove the Pigeon Hole Principle. The amount of type-in for this theorem is relatively large. The reason is that the theorem is about concepts not already in the system's data base and about which the system knows nothing. Thus, we here have to build up a fair number of facts.

```
21.  Definition.
     (ALL.DISTINCT L)
        =
     (IF (NLISTP L)
         T
         (AND (NOT (MEMBER (CAR L) (CDR L)))
              (ALL.DISTINCT (CDR L))))

22.  Definition.
     (ALL.LESSEQP L N)
        =
     (IF (NLISTP L)
         T
         (AND (LEQ (CAR L) N)
              (ALL.LESSEQP (CDR L) N)))

23.  Definition.
     (ALL.NON.ZEROP L)
        =
     (IF (NLISTP L)
         T
         (AND (NOT (ZEROP (CAR L)))
              (ALL.NON.ZEROP (CDR L))))

24.  Definition.
     (POSITIVES N)
        =
     (IF (ZEROP N)
         NIL
         (CONS N (POSITIVES (SUB1 N))))
```

```
25.  Theorem.  LISTP.POSITIVES (rewrite):
     (EQUAL (LISTP (POSITIVES N))
            (NOT (ZEROP N)))


26.  Theorem.  CAR.POSITIVES (rewrite):
     (EQUAL (CAR (POSITIVES N))
            (IF (ZEROP N) 0 N))

27.  Theorem.  MEMBER.POSITIVES (rewrite):
     (EQUAL (MEMBER X (POSITIVES N))
            (IF (ZEROP X) F (LESSP X (ADD1 N))))

28.  Theorem.  ALL.NON.ZEROP.DELETE (rewrite):
     (IMPLIES (ALL.NON.ZEROP L)
              (ALL.NON.ZEROP (DELETE X L)))

29.  Theorem.  ALL.DISTINCT.DELETE (rewrite):
     (IMPLIES (ALL.DISTINCT L)
              (ALL.DISTINCT (DELETE X L)))

30.  Theorem.  PIGEON.HOLE.PRINCIPLE/LEMMA.1 (rewrite):
     (IMPLIES (AND (ALL.DISTINCT L)
                   (ALL.LESSEQP L (ADD1 N)))
              (ALL.LESSEQP (DELETE (ADD1 N) L) N))

31.  Theorem.  PIGEON.HOLE.PRINCIPLE/LEMMA.2 (rewrite):
     (IMPLIES (AND (NOT (MEMBER (ADD1 N) X))
                   (ALL.LESSEQP X (ADD1 N)))
              (ALL.LESSEQP X N))

32.  Theorem.  PERM.MEMBER (rewrite):
     (IMPLIES (AND (PERM A B) (MEMBER X A))
              (MEMBER X B))
```

The proof of the Pigeon Hole Principle we give employs an induction argument the system does not automatically construct. To tell it the induction argument we want used, we define a recursive function that mirrors the induction. The proof of the well-foundedness of the recursion justifies the induction scheme suggested by the function. Our first mechanical proof of the Pigeon Hole Principle used a machine generated induction, but required more preliminary work in the form of lemmas about ALL.LESSEQP, DELETE, and ALL.DISTINCT.

```
33.  Definition.
     (PIGEON.HOLE.INDUCTION L)
        =
     (IF (LISTP L)
         (IF (MEMBER (LENGTH L) L)
             (PIGEON.HOLE.INDUCTION (DELETE (LENGTH L) L))
             (PIGEON.HOLE.INDUCTION (CDR L)))
         T)
```

```
34.  Theorem.  PIGEON.HOLE.PRINCIPLE:
     (IMPLIES (AND (ALL.NON.ZEROP L)
                   (ALL.DISTINCT L)
                   (ALL.LESSEQP L (LENGTH L)))
              (PERM (POSITIVES (LENGTH L)) L))
     Hint:  Induct as for (PIGEON.HOLE.INDUCTION L).
```

We conclude this subsection by anticipating our use of the Pigeon Hole Principle by proving some elegant relations between permutations, products, the positives and the factorial function.

```
35.  Theorem.  PERM.TIMES.LIST:
     (IMPLIES (PERM L1 L2)
              (EQUAL (TIMES.LIST L1)
                     (TIMES.LIST L2)))

36.  Theorem.  TIMES.LIST.POSITIVES (rewrite):
     (EQUAL (TIMES.LIST (POSITIVES N))
            (FACT N))

37.  Theorem.  TIMES.LIST.EQUAL.FACT (rewrite):
     (IMPLIES (PERM (POSITIVES N) L)
              (EQUAL (TIMES.LIST L) (FACT N)))
     Hint:  Consider:
            PERM.TIMES.LIST with {L1/(POSITIVES N), L2/L}

38.  Theorem.  PRIME.FACT (rewrite):
     (IMPLIES (AND (PRIME P) (LESSP N P))
              (NOT (EQUAL (REMAINDER (FACT N) P) 0)))
     Hint:  Induct as for (FACT N).
```

8.5. Fermat's Theorem

This subsection is the formalization of the proof in Box 2.

```
39.  Definition.
     (S N M P)
        =
     (IF (ZEROP N)
         NIL
         (CONS (REMAINDER (TIMES M N) P)
               (S (SUB1 N) M P)))

40.  Theorem.  REMAINDER.TIMES.LIST.S:
     (EQUAL (REMAINDER (TIMES.LIST (S N M P)) P)
            (REMAINDER (TIMES (FACT N) (EXP M N))
                       P))

41.  Theorem.  ALL.DISTINCT.S/LEMMA (rewrite):
     (IMPLIES (AND (PRIME P)
                   (NOT (EQUAL (REMAINDER M P) 0))
                   (NUMBERP N1)
                   (LESSP N2 N1)
                   (LESSP N1 P))
              (NOT (MEMBER (REMAINDER (TIMES M N1) P)
                           (S N2 M P))))
     Hint:  Induct as for (S N2 M P).
```

```
42.  Theorem.  ALL.DISTINCT.S (rewrite):
     (IMPLIES (AND (PRIME P)
                   (NOT (EQUAL (REMAINDER M P) 0))
                   (LESSP N P))
              (ALL.DISTINCT (S N M P)))

43.  Theorem.  ALL.NON.ZEROP.S (rewrite):
     (IMPLIES (AND (PRIME P)
                   (NOT (EQUAL (REMAINDER M P) 0))
                   (LESSP N P))
              (ALL.NON.ZEROP (S N M P)))

44.  Theorem.  ALL.LESSEQP.S (rewrite):
     (IMPLIES (NOT (ZEROP P))
              (ALL.LESSEQP (S N M P) (SUB1 P)))

45.  Theorem.  LENGTH.S (rewrite):
     (EQUAL (LENGTH (S N M P)) (FIX N))

46.  Theorem.  FERMAT.THM (rewrite):
     (IMPLIES (AND (PRIME P)
                   (NOT (EQUAL (REMAINDER M P) 0)))
              (EQUAL (REMAINDER (EXP M (SUB1 P)) P)
                     1))
     Hints:  Consider:
             PIGEON.HOLE.PRINCIPLE with {L/(S (SUB1 P) M P)}
             REMAINDER.TIMES.LIST.S with {N/(SUB1 P)}
```

8.6. Invertibility of CRYPT

This subsection is the formalization of the proof in Box 3.

```
47.  Theorem.  CRYPT.INVERTS/STEP.1:
     (IMPLIES
      (PRIME P)
      (EQUAL (REMAINDER (TIMES M (EXP M (TIMES K (SUB1 P))))
                        P)
             (REMAINDER M P)))

48.  Theorem.  CRYPT.INVERTS/STEP.1A (rewrite):
     (IMPLIES
      (PRIME P)
      (EQUAL
       (REMAINDER
                 (TIMES M
                        (EXP M (TIMES K (SUB1 P) (SUB1 Q))))
                 P)
       (REMAINDER M P)))
     Hint:  Consider:
            CRYPT.INVERTS/STEP.1 with {K/(TIMES K (SUB1 Q))}
```

```
49.  Theorem.  CRYPT.INVERTS/STEP.1B (rewrite):
     (IMPLIES
      (PRIME Q)
      (EQUAL
       (REMAINDER
                 (TIMES M
                        (EXP M (TIMES K (SUB1 P) (SUB1 Q))))
                 Q)
       (REMAINDER M Q)))
     Hint:  Consider:
            CRYPT.INVERTS/STEP.1
             with {P/Q, K/(TIMES K (SUB1 P))}


50.  Theorem.  CRYPT.INVERTS/STEP.2 (rewrite):
     (IMPLIES
            (AND (PRIME P)
                 (PRIME Q)
                 (NOT (EQUAL P Q))
                 (NUMBERP M)
                 (LESSP M (TIMES P Q))
                 (EQUAL (REMAINDER ED (TIMES (SUB1 P) (SUB1 Q)))
                        1))
            (EQUAL (REMAINDER (EXP M ED) (TIMES P Q))
                   M))

51.  Theorem.  CRYPT.INVERTS:
     (IMPLIES
           (AND (PRIME P)
                (PRIME Q)
                (NOT (EQUAL P Q))
                (EQUAL N (TIMES P Q))
                (NUMBERP M)
                (LESSP M N)
                (EQUAL (REMAINDER (TIMES E D)
                                  (TIMES (SUB1 P) (SUB1 Q)))
                       1))
           (EQUAL (CRYPT (CRYPT M E N) D N) M))
```

References

62a. W. Hunt. FM8501: A Verified Microprocessor. LNCS 795. Springer-Verlag. 1994.

69a. N. Shankar. Metamathematics, Machines, and Goedel's Proof, Cambridge University Press, 1994.

1. N. Bourbaki. *Elements of Mathematics.* Addison Wesley, Reading, Massachusetts, 1968.

2. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

3. R. S. Boyer and J S. Moore. Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

4. R. S. Boyer and J S. Moore. A Verification Condition Generator for FORTRAN. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

5. R. S. Boyer and J S. Moore. MJRTY - A Fast Majority Vote Algorithm. Technical Report ICSCA-CMP-32, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1982.

6. R. S. Boyer and J S. Moore. "Proof Checking the RSA Public Key Encryption Algorithm". *American Mathematical Monthly 91*, 3 (1984), 181-189.

7. R. S. Boyer and J S. Moore. "A Mechanical Proof of the Unsolvability of the Halting Problem". *JACM 31*, 3 (1984), 441-458.

8. R. S. Boyer and J S. Moore. A Mechanical Proof of the Turing Completeness of Pure Lisp. In *Automated Theorem Proving: After 25 Years*, W.W. Bledsoe and D.W. Loveland, Eds., American Mathematical Society, Providence, R.I., 1984, pp. 133-167.

9. Benedetto Lorenzo Di Vito. *Verification of Communications Protocols and Abstract Process Models*. Ph.D. Th., University of Texas at Austin, 1982.

10. Richard J. Feiertag. A Technique for Proving Specifications are Multilevel Secure. Technical Report CSL-109, SRI International, 1981.

11. K. Godel. On Formally Undecidable Propositions of *Principia Mathematica* and Related Systems. In *From Frege to Godel*, J. van Heijenoort, Ed., Harvard University Press, Cambridge, Massachusetts, 1967.

12. G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers.* Oxford University Press, 1979.

13. D. E. Knuth and P. Bendix. Simple Word Problems in Universal Algebras. In *Computational Problems in Abstract Algebras*, J. Leech, Ed., Pergamon Press, Oxford, 1970, pp. 263-297.

14. D. E. Knuth. *The Art of Computer Programming. Volume 1/ Fundamental Algorithms.* Addison-Wesley Publishing Co., Reading, MA, 1973.

15. D. E. Knuth. *The Art of Computer Programming. Volume 2/ Seminumerical Algorithms.* Addison-Wesley Publishing Co., Reading, MA, 1981.

16. J S. Moore. "A Mechanical Proof of the Termination of Takeuchi's Function". *Information Processing Letters 9*, 4 (1979), 176-181.

17. R. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". *Communications of the ACM 21*, 2 (1978), 120-126.

18. David M. Russinoff. A Mechanical Proof of Wilson's Theorem. Department of Computer Sciences, University of Texas at Austin, 1983.

Table of Contents