Fast Checkers for Cryptography

Kireeti Kompella * Leonard Adleman *
Department of Computer Science
University of Southern California
Los Angeles, CA 90089-0782

1 Introduction

Fast Checkers ...

Program correctness is a serious concern, and has consequently received considerable attention. This attention has taken three approaches:

- **mathematical:** *prove* programs correct;
- **empirical:** *test* programs for bugs; and
- **engineering:** *design* programs well.

While considerable progress has been made, the question of whether a given computation was performed correctly still has no practical solution. However, a new approach, proposed by Manuel Blum, promises to be both practical and of theoretical significance, and is intrinsically closer to a computer scientist's heart, since the approach is

- **algorithmic:** *check* every computation

using a *program checker*. Program checkers are designed for specific computational problems; a checker is then an algorithm such that, given a program that is supposed to solve that

*Research supported under NSF Grant # CCR 8519296 problem, and an input value, if the program computes correctly at that input, the checker says "CORRECT" (with a small probability of error); alternately, if the program has a bug, it says "BUGGY" (again, with a small chance of error).

Program checkers have the advantages that 1) they check entire computations (software + hardware + operating system); 2) they provide certificates on individual computations; and 3) the effort need not be duplicated if a different program for the same problem is to be checked. On the other hand, the price for increased confidence is increased computation, making *fast* checkers especially attractive.

...for Cryptography

If the average user is concerned about program correctness, how much more so the cryptographer, who, by the nature of his profession, deals with information of critical importance, making the correct manipulation of this information vital? How often is he willing to pay the price of slower speed for the sake of increased confidence? Program checkers make this choice available.

This paper describes a fast checker for modular exponentiation, the computational problem underlying RSA, one that, for modulus n, requires $O(\log\log n)$ queries and modular multiplications for a given confidence level. This paper also presents a hypothesis that implies the existence of a "constant-query" checker, requiring only a constant number of queries and modular multiplications, independent of the input. Finally, it is shown without hypothesis that in many practical cases, constant-query checkers can be obtained. Independently, Ronitt Rubinfeld [R] has devised a checker for a restricted version of modular exponentiation[1] that requires $O((\log\log n)^3)$ queries and $O((\log\log n)^4)$ modular multiplications for modulus n.

In passing, it is noted that this checker can be used in a number of cryptographic contexts, e.g., the Diffie-Hellman key exchange and discrete logarithm based systems. Furthermore, an entirely analogous checker can be used to check elliptic curve discrete logarithm systems, where exponentiation corresponds to multiplying points on the curve.

1.1 Definition of Program Checkers

For completeness, Blum's original definition of a program checker is given:

Definition 1 (Blum) *Let π be a computational problem. For x an input to π, let $\pi(x)$ denote the output of π. Call C_π a program checker for problem π, if for all programs P that halt on all inputs, for all instances I of π, and for all positive integers k (presented in unary), C_π^P is a probabilistic oracle Turing machine (with oracle P) such that:*

[1]This version requires that the number being exponentiated be relatively prime to the modulus.

1. *If $P(x) = \pi(x)$ for all instances x of π, then with probability $\geq 1 - 1/2^k$, $C_\pi^P(I;k) =$ CORRECT (i.e., $P(I)$ is correct);*

2. *If $P(I) \neq \pi(I)$ then with probability $\geq 1 - 1/2^k$, $C_\pi^P(I;k) =$ BUGGY (i.e., P has a "bug").*

Clearly, every computable problem π has a trivial checker: run a correct program for π, and check whether the given program produced the same answer. Thus, to obtain useful checkers, it will be required that the time for checking, apart from queries to the program, be $o(T_\pi)$, where T_π is the time complexity of solving π, i.e., that the checker run much faster than any program for π; such a checker will be called *fast.* If T_π is not known (as is often the case), one can aim for checkers that run in $o(U_\pi)$, where U_π is the best known upper bound for π; such checkers will be called *"fast"*. A *constant-query* checker is a fast (or "fast") checker that, for each value of the confidence parameter k, requires only a constant number of calls to the program being checked, and the arguments of each such call have size at most that of the original input. In such a case, one can transform any "reasonable" program for π to a "self-checking" program that has a running time of the same order of complexity.

2 Checking Modular Exponentiation

2.1 Problem Statement

Both RSA encryption/decryption and the Diffie-Hellman key exchange protocol have the same underlying computational problem, namely, modular exponentiation (ME), which can be described as follows:

Input: a, b, m: positive integers, with $a < m$.
Output: $c \equiv a^b \bmod m$.

The main result:

Theorem 1 (Fast Checkers for Modular Exponentiation Exist.) *There exists a program checker C_{ME} for modular exponentiation such that for all programs P which halt everywhere, and all inputs $a, b, m \in \mathbb{Z}_{>0}$ with $a < m$,*

1. *C_{ME}^P makes $O(\log\log b)$ queries of the program P;*

2. *C_{ME}^P requires $O(\log\log b + \log\log m)$ multiplications modulo m.*

2.2 Remarks

The checker C_{ME} is based on the *tester-checker* paradigm, introduced in [AK]. That is, it follows a two-stage protocol, where, in the first stage, the checker *tests* that the program satisfies some statistical property (e.g., correctness on a certain fraction of the inputs); and in the second, it *checks* the program on the given input, making use of the property just verified.

To see whether C_{ME} is a *fast* checker, one has to compare the time to check against the time to compute. The fastest known algorithm for modular exponentiation takes $O(\log b)$ modular multiplications for exponent b. Thus, until an algorithm that takes $O(\log\log b)$ multiplications is invented, the given checker can be deemed fast. Independently, Ronitt Rubinfeld [R] discovered a checker that requires $O(\log\log b \cdot \log\log^3 m)$ modular multiplications and $O(\log\log b \log\log^2 m)$ calls to the program.

The checker C_{ME} can easily be modified to check exponentiation on any group, and in many semi-groups, again with $O(\log\log n)$ queries and group multiplications, where n is the exponent. Thus, for example, one can obtain fast checkers for exponentiation in polynomial rings and on elliptic curve groups.

2.3 Informal Description

(To simplify this description, the modulus m is assumed to be prime. This restriction is not required for the actual checker.)

At the heart of the checker for modular exponentiation lies the familiar algebraic identity:

$$a^e \cdot a^f \equiv a^{e+f} \bmod m; \tag{1}$$

Let P be a program that purports to perform modular exponentiation, and let (a, b, m) be the given input. Suppose that, for base a and modulus m, P exponentiates correctly at sufficiently many exponents f, i.e.,

$$P(a, f, m) \equiv a^f \bmod m \text{ for at least } 5/6 \text{ of } f \in \{0, \ldots, B\} \tag{2}$$

(where B is appropriately chosen), but, at the given input,

$$P(a, b, m) \not\equiv a^b \bmod m. \tag{3}$$

Then identity (1) suggests the following check: pick f randomly from $\{0, \ldots, B\}$, and check whether $P(a, b, m) \cdot P(a, f, m) \equiv P(a, b+f, m) \bmod m$. If f is such that $P(a, f, m) \equiv a^f \bmod m$ and $P(a, b + f, m) \equiv a^{b+f} \bmod m$, then (3) implies that the check will fail. Moreover, (2) implies that picking such an f is reasonably likely (provided that $b + f \leq B$). Thus, the tester-checker paradigm suggests itself: first, obtain confidence that the program P satisfies (2), then use identity (1) as outlined to obtain confidence that $P(a, b, m) \equiv a^b \bmod m$.

However, the "tester" phase of verifying that (2) holds needs to be performed carefully. Direct computation is clearly too slow, taking $O(\log b)$ multiplications, as opposed to the desired $O(\log\log b)$ multiplications. Thus, a more subtle approach is required: establish (2) for a much smaller interval, then inductively extend to larger intervals, using P itself to perform the checks.

For the inductive step, the identity used is the following:

$$a^{h\cdot 2^{2^j}+l} \equiv (a^h)^{2^{2^j}} \cdot a^l \bmod m$$

Thus, checking on exponents 2^{j+1} bits long (namely, $h \cdot 2^{2^j} + l$, with h and l of size 2^j) is reduced to checking on exponents of half this size (namely, h and l). However, one more check is needed: that $x^{2^{2^j}} \bmod m$ is computed correctly for arbitrary x. Again, the tester-checker paradigm is brought into play: first test that $P(y, 2^{2^j}, m) \equiv y^{2^{2^j}} \bmod m$, for most $y \in \{1, \ldots, m-1\}$, then use the identity

$$(xy)^e \equiv x^e \cdot y^e \bmod m$$

to check that $x^{2^{2^j}} \bmod m$ is computed correctly. Once again, the testing is done inductively: if P is tested for 2^{2^j}, then it can be tested for $2^{2^{j+1}}$ using:

$$y^{2^{2^{j+1}}} \equiv (y^{2^{2^j}})^{2^{2^j}} \bmod m.$$

Now consider the case when m is not prime. Then cancellation is no longer valid in general, and this introduces two problems: first, even if a "nice" f is picked, the check suggested by (1) may not work; and second, the check for $P(x, 2^{2^j}, m)$ also may not work. The former problem is circumvented by working modulo the largest factor of m relatively prime to a (without explicitly computing it), so that cancellation is reinstated; and the latter is solved by using a "generalized" cancellation law modulo m:

$$z(y^e) \equiv (xy)^e \ \&\ z(y+1)^e \equiv (x(y+1))^e \text{ implies } z \equiv x^e.$$

2.4 The Checker C_{ME}

The checker is presented below in detail. Note that the modulus m is no longer restricted to be prime.

The following notation will be used: for any positive integers m and a, write m as a product $\prod p_i^{e_i}$ of distinct primes, and denote by $m[a]$ the product $\prod_{p_i \mid a} p_i^{e_i}$, and let $\overline{m[a]} = m/m[a]$. Note that $(m[a], \overline{m[a]}) = (a, \overline{m[a]}) = 1$. Let $exp(m) = max_i\{e_i\}$; observe that $exp(m) \leq \log m$.

For $j \in \mathbb{Z}_{\geq 0}$, let $T_1(P, j, m)$ be the property that $\frac{\#\{y|0<y<m \& P(y,2^{2^j},m)\equiv y^{2^{2^j}} \bmod m\}}{\#\{y|0<y<m\}} \geq \frac{7}{8}$, and $T_2(P, a, j, m)$ that $\frac{\#\{f|0\leq f<2^{2^j} \& P(a,f,m)\equiv a^f \bmod \overline{m[a]}\}}{\#\{f|0\leq f<2^{2^j}\}} \geq \frac{5}{6}$. The checker first attempts to verify that $T_1(P, j, m)$ and $T_2(P, a, j, m)$ hold for $0 \leq j \leq \log\log b$. It then checks the program on the given input.

$C^P_{ME}(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{m}; \boldsymbol{k})$:

Input: P: a program (supposedly for modular exponentiation) that halts on all inputs;
a, b, m: positive integers such that $0 < a < m$;
k: confidence parameter (given in unary).
Output: 'CORRECT' if $P(A, B, M) \equiv A^B \bmod M$ for all $A, B, M \in \mathbb{Z}_{>0}$ with $A < M$;
'BUGGY' (with probability $\geq 1 - 1/2^k$) if $P(a, b, m) \not\equiv a^b \bmod m$.

```
begin
    Set $c = exp(m)$, if known; else set $c = \lceil \log m \rceil$
    (* For small exponents $b \leq c$, check by direct computation. *)
    if $b \leq exp(m)$, compute $a^b \bmod m$ directly;
        if $P(a, b, m) \not\equiv a^b \bmod m$, output 'BUGGY'
        otherwise, output 'CORRECT';
        halt.

    set $n = \lceil \log\log b \rceil$.
(* Tester stage *)
    test1($n, m, k$)
    test2($a, n, m, k$)

(* Checker stage: establish correctness at given input *)
    do $k$ times
        check2($a, b, n, m$) (* Check that $P(a, b, m) \equiv a^b \bmod \overline{m[a]}$ *)
    (* Then, that $P(a, b, m) \equiv a^b \bmod m[a]$ *)
    directly compute $a^c \bmod m$
    if $P(a, b - c, m) \cdot a^c \not\equiv P(a, b, m) \bmod m$ then output 'BUGGY'
    else output 'CORRECT'
end
```

```
test1($n, m, k$):
(* Test whether $T_1(P, j, m)$ holds for $0 \leq j \leq n$. *)
   begin
        (* The base case $j = 0$ is just squaring: test directly *)
             do $6k$ times
                    pick $x$ randomly from $\{1, \ldots, m - 1\}$
```

```
                if $P(x,2,m) \not\equiv x^2$ mod $m$ then output 'BUGGY' and halt

      for $j \in \{1,\ldots,n\}$ do
            do $11k$ times:
                pick $x$ randomly from $\{1,\ldots,m-1\}$
                **check1**$(x,j-1,m)$; let $y = P(x,2^{2^{j-1}},m)$
                **check1**$(y,j-1,m)$; let $z = P(y,2^{2^{j-1}},m)$
                if $P(x,2^{2^j},m) \not\equiv z$ mod $m$ then output 'BUGGY' and halt
  **end**

**check1**$(x,j,m)$:
(* check that $P(x,2^{2^j},m) \equiv x^{2^{2^j}}$ mod $m$, given that $T_1(P,j,m)$ holds. *)
  **begin**
      pick $y$ randomly from $\{1,\ldots,m-1\}$
      if $P(x,2^{2^j},m) \cdot P(y,2^{2^j},m) \not\equiv P(x \cdot y \bmod m, 2^{2^j},m)$ mod $m$ then
            output 'BUGGY' and halt.
      if $P(x,2^{2^j},m) \cdot P(y+1,2^{2^j},m) \not\equiv P(x \cdot (y+1) \bmod m, 2^{2^j},m)$ mod $m$ then
            output 'BUGGY' and halt.
  **end**

**test2**$(a,n,m,k)$:
(* Test whether $T_2(P,a,j,m)$ holds for $0 \le j \le n$. *)
  **begin**
      (* The base case $j = 0$ is checked directly. *)
      if $P(a,0,m) \not\equiv 1$ mod $m$ or $P(a,1,m) \not\equiv a$ mod $m$ then
            output 'BUGGY,' and halt.

      for $j \in \{1,\ldots,n\}$
            do $8k$ times
                pick $e$ randomly from $\{0,\ldots,2^{2^j}-1\}$
                write $e = h \cdot 2^{2^{j-1}} + l$, with $0 \le h,l < 2^{2^{j-1}}$
                **check2**$(a,h,j-1,m)$; let $x = P(a,h,m)$
                **check1**$(x,j-1,m)$;   let $y = P(x,2^{2^{j-1}},m)$
                **check2**$(a,l,j-1,m)$; let $z = P(a,l,m)$
                if $y \cdot z \not\equiv P(a,e,m)$ mod $m$ then output 'BUGGY' and halt
  **end**

**check2**$(a,e,j,m)$:
(* check that $P(a,e,m) \equiv a^e$ mod $\overline{m[a]}$, given that $e < 2^{2^j}$ and $T_2(P,a,j,m)$ holds. *)
  **begin**
      pick $f$ randomly from $\{0,\ldots,2^{2^j}-1\}$
```

```
        if $f > e$ then
            if $P(a,e,m) \cdot P(a,f-e,m) \not\equiv P(a,f,m) \bmod m$ then
                output 'BUGGY' and halt.
        else
            if $P(a,f,m) \cdot P(a,e-f,m) \not\equiv P(a,e,m) \bmod m$ then
                output 'BUGGY' and halt.
    end
```

2.5 Formal Proof of Theorem 1

In this section, Theorem 1 is proved. The proof proceeds via several lemmas that assert that the tests and checks given perform as desired:

Lemma 1 (check1 works) *For all $j \in \mathbb{Z}_{\geq 0}$, for all $x, m \in \mathbb{Z}_{>0}$ with $x < m$,*

1. *If $P(y, 2^{2^j}, m) \equiv y^{2^{2^j}} \bmod m$ for all $0 < y < m$, then* **check1***(x,j,m) does nothing.*
2. *If $T_1(P,j,m)$ holds, but $P(x,j,m) \not\equiv x^{2^{2^j}} \bmod m$, then with probability $\geq 1/2$* **check1***(x,j,m) outputs 'BUGGY' and halts.*

Proof: 1) Straightforward.

2) Assume the antecedent. For $0 < y < m$, let $S_y = \{y, xy \bmod m, y+1, x(y+1) \bmod m\}$; call y *bad* if S_y has a z such that $P(z, 2^{2^j}, m) \not\equiv z^{2^{2^j}} \bmod m$. Now, each z with $0 < z < m$ can appear in at most 4 S_y's; and since $\#\{z | 0 < z < m \ \& \ P(z, 2^{2^j}, m) \not\equiv z^{2^{2^j}} \bmod m\} < \frac{m-1}{8}$, there are less than $\frac{m-1}{2}$ bad y's. But if y is not bad, then **check1** will output 'BUGGY' and halt, for otherwise, one must have

$$P(x, 2^{2^j}, m) \cdot y^{2^{2^j}} \equiv (xy)^{2^{2^j}} \equiv x^{2^{2^j}} \cdot y^{2^{2^j}} \bmod \overline{m[y]}$$

(since $\overline{m[y]} | m$), so that $P(x, 2^{2^j}, m) \equiv x^{2^{2^j}} \bmod \overline{m[y]}$ (since $(y, \overline{m[y]}) = 1$, one can cancel); and similarly, $P(x, 2^{2^j}, m) \boxminus x^{2^{2^j}} \bmod m[y]$ (since $m[y] | m$, and $(y+1, m[y]) = 1$), whence $P(x, 2^{2^j}, m) \equiv x^{2^{2^j}} \bmod m$ by the Chinese Remainder Theorem, a contradiction. Thus, with probability at least 1/2, check1(x,j,m) will output 'BUGGY' and halt. □

Lemma 2 (test1 works) *For all $n \in \mathbb{Z}_{\geq 0}$, for all $m, k \in \mathbb{Z}_{>0}$,*

1. *If $P(y, 2^{2^j}, m) \equiv y^{2^{2^j}} \bmod m$ for $0 < y < m$, then* **test1***(n,m,k) does nothing.*
2. *If $T_2(P,a,j,m)$ fails to hold for some $0 \leq j \leq n$, then with probability at least $1 - 2^{-k}$,* **test1***(n,m,k) will output 'BUGGY' and halt.*

Proof: 1) Straightforward.

2) Let j_0 be the smallest j such that $T_2(P,a,j,m)$ fails to hold. If $j_0 = 0$, then P squares incorrectly at least 1/8 of the time, and in 6 passes through the first loop, an instance where $P(x,2,m) \not\equiv x^2 \bmod m$ will be found with probability at least 1/2. Therefore, at the end of the loop, with probability $\geq 1 - 2^{-k}$, **test1** will output 'BUGGY' and halt.

Now suppose $j_0 > 0$. For random x, $\text{Prob}[P(x, 2^{2^{j_0}}, m) \not\equiv x^{2^{2^{j_0}}} \bmod m] \geq 1/8$. For such an x, if $P(x, 2^{2^{j_0-1}}, m) \equiv x^{2^{2^{j_0-1}}} \bmod m$, and $P(x^{2^{2^{j_0-1}}}, 2^{2^{j_0-1}}, m) \equiv (x^{2^{2^{j_0-1}}})^{2^{2^{j_0-1}}} \equiv x^{2^{2^{j_0}}} \bmod m$, then **test1** will output 'BUGGY' and halt. If either of these values is computed incorrectly, then with probability at least 1/2, **check1** will output 'BUGGY' and halt. So, every pass through the second loop will detect a bug with probability at least 1/16. Thus, at the end of the loop, with probability $\geq 1 - 2^{-k}$, **test1** will output 'BUGGY' and halt. □

Lemma 3 (check2 works) *For all* $e, j \in \mathbb{Z}_{\geq 0}$ *with* $e < 2^{2^j}$, *for all* $a, m \in \mathbb{Z}_{>0}$ *with* $a < m$,

1. *If* $P(a,f,m) \equiv a^f \bmod m$ *for all* $0 \leq f < 2^{2^j}$, *then* **check2**(a,e,j,m) *does nothing.*
2. *If* $T_1(P,j,m)$ *and* $T_2(P,a,j,m)$ *hold, but* $P(a,e,m) \not\equiv a^e \bmod \overline{m[a]}$ *then with probability* $\geq 1/2$ **check2**(a,e,j,m) *outputs 'BUGGY' and halts.*

Proof: 1) Straightforward.

2) Assume the antecedent. Arguing as in lemma 1, one can show that the probability of picking f such that, modulo $\overline{m[a]}$,

$$P(a,f,m) \equiv a^f, P(a,e-f,m) \equiv a^{e-f} \text{ (if } e \geq f\text{), and } P(a,f-e,m) \equiv a^{f-e} \text{ (if } f \geq e\text{)}$$

is at least 1/2. But for such an f (say $f \geq e$), $P(a,e,m) \cdot P(a,f-e,m) \not\equiv P(a,f,m) \bmod \overline{m[a]}$, since $P(a,e,m) \not\equiv a^e \bmod \overline{m[a]}$ and $(\overline{m[a]}, a) = 1$); thus, *a forteriori*, the congruence cannot hold mod m. Therefore, with probability at least 1/2, **check2** will output 'BUGGY' and halt. □

Lemma 4 (test2 works) *For all* $n \in \mathbb{Z}_{\geq 0}$, *for all* $a, m, k \in \mathbb{Z}_{>0}$,

1. *If* $P(a,f,m) \equiv a^f \bmod m$ *for* $0 \leq f < 2^{2^n}$, *then* **test1**(n,m,k) *does nothing.*
2. *If* $T_1(P,j,m)$ *holds for all* $0 \leq j \leq n$, *and* $T_2(P,a,j,m)$ *fails to hold for some* $0 \leq j \leq n$, *then with probability at least* $1 - 2^{-k}$, **test2**(a,n,m,k) *will output 'BUGGY' and halt.*

Proof: 1) Straightforward.

2) Let j_0 be the smallest j such that $T_2(P,a,j,m)$ fails to hold. WLOG, assume $j_0 > 0$. Then, for random $e < 2^{2^{j_0}}$, $\text{Prob}[P(a,e,m) \not\equiv a^e \bmod \overline{m[a]}] \geq 1/6$. If $x := P(a,h,m) \equiv$

$a^h \bmod \overline{m[a]}$, $P(x, 2^{2^{j_0-1}}, m) \equiv x^{2^{2^{j_0-1}}} \bmod m$ (and thus mod $\overline{m[a]}$), and $P(a, l, m) \equiv a^l \bmod \overline{m[a]}$, then **test2** will output 'BUGGY' and halt. If any of the congruences fails, then the corresponding check will catch the bug with probability at least 1/2. Therefore, for random e, Prob[**test2** finds a bug]$\geq 1/12$. Thus, after $11k$ choices of e, Prob[**test2** says 'BUGGY'] $\geq 1 - 2^{-k}$. □

Proof of Theorem 1:
To show that C_{ME} is in fact a checker for modular exponentiation, observe first that for any correct program P, C_{ME}^P answers 'CORRECT' on all valid inputs. On the other hand, if P and $< a, b, m >$ are such that $P(a, b, m) \not\equiv a^b \bmod m$, then one of the following must hold (let $n = \lceil \log \log b \rceil$, and assume WLOG that $b \geq c = exp(m)$):

1. $T_1(P, j, m)$ fails to hold for some $0 \leq j \leq n$;
2. $T_1(P, j, m)$ holds for all $0 \leq j \leq n$, but $T_2(P, a, j, m)$ fails to hold for some $0 \leq j \leq n$;
3. $T_1(P, j, m)$ and $T_2(P, a, j, m)$ hold for all $0 \leq j \leq n$, but $P(a, b, m) \not\equiv a^b \bmod \overline{m[a]}$;
4. $T_1(P, j, m)$ and $T_2(P, a, j, m)$ hold for all $0 \leq j \leq n$, but $P(a, b, m) \not\equiv a^b \bmod m[a]$.

Case 1: Lemma 2 applies;
Case 2: Lemma 4 applies;
Case 3: Lemma 3 applies;
Case 4: For $b \geq c$, $a^b \equiv 0 \bmod m[a]$. Thus, $P(a, b - c, m) \cdot a^c \equiv 0 \bmod m[a]$. Hence, if $P(a, b, m) \not\equiv a^b \bmod m[a]$, then $P(a, b, m) \not\equiv P(a, b - c, m) \cdot a^c \bmod m[a]$, so the congruence fails mod m. Thus, in any case, $C_{ME}^P(a, b, m; k)$ will output 'BUGGY' with probability at least $1 - 2^{-k}$.

Verifying the running time is a straightforward task. □

Remark: when $exp(m)$ is bounded, C_{ME} requires only $O(\log \log b)$ queries and modular multiplications: checking for small exponents is done explicitly, and not by direct computation. This is the case when using the Diffie-Hellman key exchange protocol ($exp(m) = 1$ since the modulus is prime), and for the RSA cryptosystem (the modulus is usually square- or cube-free.)

2.6 Constant-Query Checkers for Modular Exponentiation

While C_{ME} is a "fast" (with respect to the current best upper bounds) checker, one can ask whether there exists a constant-query checker for modular exponentiation, i.e., a checker that requires a constant number of queries and modular multiplications. In this section, it is shown that, under a hypothesis, such a checker does indeed exist for what one might call the RSA problem: on input positive integers x, y, z with $x < z$ and $(x, z) = 1$, compute $x^y \bmod z$.

The hypothesis is now presented. Some definitions are required:

Definition 2 *For all $n \in \mathbf{Z}_{>0}$, for all $f : \mathbf{Z} \to \mathbf{Z}$, for all $\gamma \in \mathbf{R}$ with $0 \leq \gamma \leq 1$,*

1. *f is γ-*homomorphic for *n iff*
$$\frac{\#\{x \in \mathbf{Z} | 0 \leq x < n \ \&\ f(x) \not\equiv 0 \bmod n\}}{\#\{x \in \mathbf{Z} | 0 \leq x < n\}} \geq \gamma$$
and
$$\frac{\#\{x, y \in \mathbf{Z} \mid 0 \leq x, y < n \ \&\ f(x+y) \equiv f(x) \cdot f(y) \bmod n\}}{\#\{x, y \in \mathbf{Z} | 0 \leq x, y < n\}} \geq \gamma$$
and
$$\frac{\#\{x \in \mathbf{Z} \mid 0 \leq x < n \ \&\ f(x+1) \equiv f(x) \cdot f(1) \bmod n\}}{\#\{x \in \mathbf{Z} | 0 \leq x < n\}} \geq \gamma$$

2. *f is γ-*exponential for *n iff*
$$\frac{\#\{x \in \mathbf{Z} | 0 \leq x < n \ \&\ f(x) \equiv f(1)^x \bmod n\}}{\#\{x \in \mathbf{Z} | 0 \leq x < n\}} \geq \gamma$$

Hypothesis 1 *There exists $\gamma \in \mathbf{R}$ with $0 \leq \gamma < 1$ such that for all $n \in \mathbf{Z}_{>0}$ and for all $f : \mathbf{Z} \to \mathbf{Z}$, if f is γ-homomorphic for n, then f is $\frac{9}{10}$-exponential for n.*

In support of the hypothesis, note the following lemma (proof omitted):

Lemma 5 *For all $n \in \mathbf{Z}_{>0}$, for all $f : \mathbf{Z}/\phi(n)\mathbf{Z} \to \mathbf{Z}/n\mathbf{Z}$, f is 1-homomorphic for n iff f is 1-exponential for n.*

Further, Don Coppersmith has shown that a similar hypothesis holds when the order of the multiplicative group $\phi(n)$ is known. Using this, Blum, Luby and Rubinfeld have demonstrated constant-query checkers for modular exponentiation when n is prime, or of known factorization ([BLR]).

Theorem 2 *Hypothesis 1 implies that there exists an RSA checker C_{RSA} such that for all programs P that halt on all inputs, and all instances $x, y, z \in \mathbf{Z}_{>0}$ with $x, y < z$ and $(x, z) = 1$ and all $k \in \mathbf{Z}_{>0}$,*

1. *$C^P(x, y, z; k)$ requires at most $O(k)$ queries to P;*
2. *$C^P(x, y, z; k)$ requires at most $O(k)$ multiplications mod z.*

First the checker is presented:

Given a program P that halts on all inputs (and supposedly computes RSA), and given $x, y, z \in \mathbf{Z}_{>0}$ with $x, y < z$ and $(x, z) = 1$; $C^P_{RSA}(x, y, z; k)$ runs as follows:

```
$C^P_{RSA}(x, y, z; k)$:
Let $t_1 = \lceil -k \log_\gamma 2 \rceil$, and $t_2 = \lceil -k / \log_{4/5} 2 \rceil$, where $\gamma$ is as in Hypothesis 1.
begin
    (Ensure that $f(1) \equiv x \bmod z$.)
    if $f(1) \not\equiv x \bmod z$, output 'BUGGY' and halt.

    (Establish $\gamma$-homomorphism for $f(s) = P(x, s, z)$.)
    repeat $t_1$ times:
        choose random $i \in \mathbb{Z}$ with $0 \leq i < z$.
        if $P(x, i, z) \equiv 0 \bmod z$, output 'BUGGY' and halt.
        choose random $i, j \in \mathbb{Z}$ with $0 \leq i, j < z$.
        if $P(x, i, z) \cdot P(x, j, z) \not\equiv P(x, i + j, z) \bmod z$, output 'BUGGY' and halt.
        choose random $i \in \mathbb{Z}$ with $0 \leq i < z$.
        if $P(x, i, z) \cdot P(x, 1, z) \not\equiv P(x, i + 1, z) \bmod z$, output 'BUGGY' and halt.

    (Establish correctness on given input.)
    repeat $t_2$ times:
        choose random $r \in \mathbb{Z}$ with $0 \leq r < z$.
        if $P(x, y, z) \cdot P(x, r, z) \not\equiv P(x, y + r, z) \bmod z$, output 'BUGGY' and halt.

    Output 'CORRECT'.
end
```

Proof of Theorem 2:

Clearly, if $P(x, y, z) \equiv x^y \bmod z$ for all $x, y, z \in \mathbb{Z}_{>0}$, C_{RSA} outputs 'CORRECT'. It remains to verify that if $P(x, y, z) \not\equiv x^y \bmod z$, then $C^P_{RSA}(x, y, z; k)$ outputs 'BUGGY' with probability $\geq 1 - 1/2^k$.

Well, if f (as defined in Step 2) is not γ-homomorphic for z, then with probability $\geq 1 - 1/2^k$ (by the choice of t_1), C_{RSA} will find a bug. So, assume f is in fact γ-homomorphic. By Hypothesis 1, f is $\frac{9}{10}$-exponential. But then, for all $w \in \mathbb{Z}$ with $0 \leq w < z$,

$$\frac{\#\{r \in \mathbb{Z} \mid 0 \leq r < z \,\&\, f(r) \equiv f(1)^r \bmod z \,\&\, f(w + r) \equiv f(1)^{w+r} \bmod z\}}{\#\{r \in \mathbb{Z} \mid 0 \leq r < z\}} \geq \frac{4}{5}.$$

Thus, by the choice of t_2, the probability that C_{RSA} picks an r such that $f(r) \equiv f(1)^r \bmod z$, and $f(y + r) \equiv f(1)^{y+r} \bmod z$) is $\geq 1 - 1/2^k$. If such an r is picked, then $f(r) \cdot f(y) \not\equiv f(y + r) \bmod z$, since $f(1) \equiv x \bmod z$ by Step 1, and $(x, z) = 1$. □

2.7 "Offline" Checking

In many situations, one has the situation where the modulus is fixed. Such is the case when two parties use the RSA cryptosystem to converse. In this case, checking can be done much more efficiently, without hypothesis, by performing the tester phase offline, i.e., the program to be checked is tested to see that it works reasonably well for a given modulus. Then, whenever the program is run, the output is checked using a checker that only guarantees correctness when used with tested programs.

A checker for tested programs is now presented. First, some definitions are needed:

Definition 3 *For all $m \in \mathbb{Z}_{>0}$, for all programs P, P is m-*tested *iff*

$$\frac{\#\{x, y \in \mathbb{Z} \mid 0 \leq x, y < m^2 \ \&\ (x, m) = 1 \ \&\ P(x, y) \equiv x^y \bmod m\}}{\#\{x, y \in \mathbb{Z} \mid 0 \leq x, y < m^2 \ \&\ (x, m) = 1\}} \geq 99/100$$

Definition 4 (after Blum) *For all $m \in \mathbb{Z}_{>0}$, call TC_{RSA} an* RSA tester-checker for modulus m, *iff for all m-tested programs P that halt on all inputs, for all $x, y \in \mathbb{Z}$ with $0 \leq x, y < m$ and $(x, m) = 1$, and for all positive integers k (presented in unary), TC^P_{RSA} is a probabilistic oracle Turing machine (with oracle P) such that:*

1. *If $P(w, v) \equiv w^v \bmod m$ for all $w, v \in \mathbb{Z}$ with $0 \leq w, v < m$ and $(w, m) = 1$, then with probability $\geq 1 - 1/2^k$, $TC^P_{RSA}(x, y; k) = CORRECT$.*
2. *If $P(x, y) \not\equiv x^y \bmod m$ then with probability $\geq 1 - 1/2^k$, $TC^P_{RSA}(x, y; k) = BUGGY$.*

For all $m \in \mathbb{Z}_{>0}$ an RSA tester-checker for modulus m is now presented. Let R be a program (purported to have the property that for all $w, v \in \mathbb{Z}$ with $0 \leq w, v < m^2$ and $(w, m) = 1$, on input w, v it outputs $w^v \bmod m$) to be checked.

First, the algorithm RSA-Tester below is run to eliminate with high probability programs which are not m-tested.

Algorithm RSA-Tester:

```
Input:     $R$: a program (supposedly for RSA) that halts on all inputs;
           $m$: a positive integer;
           $k$: confidence parameter (given in unary).
    begin
       Set $t = \lceil (k+1) \log_{100/99} 2 \rceil$.
       repeat $t$ times:
           Choose random $x, y \in \mathbb{Z}$ with $0 \leq x, y < n^2$.
           If $R(x, y) \not\equiv x^y \bmod m$, output 'FAIL' and halt.
       Output 'PASS.'
    end
```

The checker phase follows only for 'm-tested' programs, for input $x, y \in Z$, with $0 \le x, y < m$ and $(x, m) = 1$. Let fail(u, v, w, x, y, z) be the predicate

$$R(w \cdot z, v) \cdot R(u, y \cdot v) \not\equiv R(u \cdot x, y \cdot v) \cdot R(w, v) \bmod n$$

Algorithm RSA Tester-Checker

```
Input:     $R$: a $m$-tested program;
           $x, y, m$: positive integers as above;
           $k$: confidence parameter (given in unary).
Output:    'CORRECT' if $P(X, Y, m) \equiv X^Y \bmod m$ for all $X, Y \in Z_{>0}$ with $X < m$ and $(X, m)$
           'BUGGY' (with probability $\ge 1 - 1/2^k$) if $P(x, y, m) \not\equiv x^y \bmod m$.

  begin
    Set $z = R(x, y)$.
    repeat $t = \lceil (k+1) \log_{25/24} 2 \rceil$ times:
        choose random $u, v, w \in Z$, with $0 \le u, v, w < m$.
        if fail$(u, v, w, x, y, z)$ or fail$(u+1, v, w, x, y, z)$
              or fail$(u, v+1, w, x, y, z)$ or fail$(u+1, v+1, w, x, y, z)$
              or fail$(u, v, w+1, x, y, z)$ or fail$(u+1, v, w+1, x, y, z)$
              or fail$(u, v+1, w, x, y, z)$ or fail$(u+1, v+1, w+1, x, y, z)$
                    Output 'BUGGY' and halt.

    Output 'CORRECT.'
  end
```

2.8 Open Problems

An important open problem is whether there exists a constant-query checker for modular exponentiation. Such a checker could prove useful in cryptography. One direction in which to pursue this question would be to prove that Hypothesis 1 holds.

A question of greater general interest is to characterize problems with constant-query checkers. Software practitioners could use this characterization for guidance in writing "checkable" programs.

References

[AK] Leonard Adleman and Kireeti Kompella, Self-Checking RSA Programs, abstract submitted to *Advances in Cryptology* (CRYPTO 89).

[B] Manuel Blum, Designing Programs to Check Their Work, Technical Report, ISRI, UC Berkeley, Nov. 1988.

[BLR] Manuel Blum, Michael Luby, and Ronitt Rubinfeld, Self-Testing/Correcting with Applications to Numerical Problems, *Proc. of the 22nd Annual ACM Symposium on Theory of Computing*, 1990, pp. 73-83.

[DH] Walt Diffie and Martin Hellman, New Directions in Cryptography, *IEEE Transcations on Information Theory*, vol. IT-22, 1976, pp. 644-654.

[RSA] Ron Rivest, Adi Shamir, and Leonard Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, *Communications of the ACM*, vol. 21, no. 2, February 1978, pp. 120-126.

[R] Ronitt Rubinfeld, private communication.