

## Checking and correcting: fingerprinting

### Problem

Check the identity of two large files in a situation where comparing them bit by bit is not feasible.

This problem arises for example when we want to check whether

- (i) a large file has been transmitted correctly over a possibly noisy channel,
- (ii) the current and an older version of a file are the same.

### Solution

Compare *fingerprints* of the files, not the whole file

Idea: the fingerprint is considerable smaller than the file itself, while distinct files have distinct fingerprints with high probability.

## Checking and correcting: fingerprinting

We assume that files are given as words over the alphabet  $\{0, 1\}$ .

### Computing a fingerprint of a word

With  $k$  understood, map a given binary word  $w = w_1 \dots w_n$  first to

$$f(w) = 2^n w_n + 2^{n-1} w_{n-1} + \dots + 2w_1,$$

and then let  $f_k(w) = f(w) \bmod k$  be the fingerprint of  $w$ .

The fingerprint  $f_k(w)$  requires approximately  $\log k$  bits of storage, where  $\log k$  is chosen much smaller than  $n$ .

The fingerprints  $f_k(u)$  and  $f_k(v)$  are the same if and only if  $f(u)$  and  $f(v)$  leave the same remainder modulo  $k$ , i.e., if

$$k \text{ divides } |f(u) - f(v)|.$$

Using a fixed value of  $k$  might be good enough for detecting random errors, but choosing  $k$  deterministically fails against deliberate changes by a malign adversary.

## Checking and correcting: fingerprinting

### Lemma

For all sufficiently large  $t$ , there are at least  $t$  prime numbers less than or equal to  $t \log t$ .

### Proof.

By the *prime number theorem*,

$$\lim_{m \rightarrow \infty} \frac{|\{p \leq m : p \text{ is prime}\}|}{m / \ln m} = 1,$$

i.e., for large  $m$ , the number of primes below  $m$  is at least  $\frac{4}{5} \frac{m}{\ln m}$ .

For  $m = t \log t$ , the number of primes below  $m$  is then at least

$$\frac{4}{5} \frac{m}{\ln m} = \frac{4}{5 \ln 2} \cdot \frac{\log t}{\log t + \log \log t} \cdot t \geq t$$

for all sufficiently large  $t$  because of  $\frac{4}{5 \ln 2} > 1$ . □

## Checking and correcting: fingerprinting

### Algorithm Fingerprint

(Parameter: a natural number  $t$ )

Input: Two words  $u$  and  $v$  of length  $n$ .

Choose  $p$  uniformly at random among the prime numbers  
below  $tn \log tn$ .

If  $f_p(u) = f_p(v)$  then accept, else reject.

A uniformly distributed prime below  $tn \log tn$  can be obtained by picking random numbers in this range and running an efficient primality test on them, where the probability of not finding a prime can be made so small that this case can be neglected.

By the lemma above, there are  $tn$  primes in this range, hence the probability of not hitting a prime when picking  $r$  numbers

is at most  $(1 - \frac{1}{r})^r \leq 1/e < 1/2$  for  $r = \log tn$ ,

and thus is at most  $2^{-k}$  for  $r = k \log tn$ .

## Checking and correcting: fingerprinting

### Proposition

Let Algorithm Fingerprint be applied to words  $u$  and  $v$  of length  $n$  where the parameter  $t$  is so large that the lemma above applies. In case both words are identical, the algorithm accepts with probability 1, in case the two words differ, the algorithm accepts with probability at most  $1/t$ .

### Proof.

If  $u$  and  $v$  are identical, then  $f_p(u) = f_p(v)$  for all values of  $p$ , hence the algorithm accepts with probability 1.

Next assume that  $u$  and  $v$  differ and let  $d = |f(u) - f(v)|$ .

The algorithm accepts if and only if  $p$  divides  $d$ .

We have  $1 \leq d \leq 2^n$ , hence  $d$  differs from 0 and has at most  $n$  distinct prime factors.

By the lemma above, there are at least  $tn$  primes below  $tn \log tn$ , thus the probability that  $p$  divides  $d$  is at most  $\frac{n}{tn} = \frac{1}{t}$ .  $\square$

## Checking and correcting: self-correcting programs

### Correctness

Existing methods for ensuring correctness of hard- or software are not fully satisfactory.

Formal methods for checking the correctness exists but often are considered to be too complicated or too complex for being applied.

The extensive runtime checks with test data that are done in practice cannot guarantee to find all errors.

### The Pentium division bug

The Pentium division bug resulted in incorrect computations for certain rare inputs (according to some sources, division was incorrect for a fraction of roughly  $10^{-10}$  of all arguments).

The division bug indicates that even the most basic functions of highly tested (and hopefully also verified) products may be incorrect, and that in particular rare errors are hard to detect.

## Checking and correcting: self-correcting programs

### Self-checking and self-correcting programs

Testing can ensure that results are correct for most inputs.

Idea: relate the computation for given inputs to the computation for inputs that are randomly chosen or are "less complex".

This idea can be used

- ▶ for checking results (i.e., for detecting errors),
- ▶ for correcting results.

Result: Programs that check or even correct themselves.

## Checking and correcting: self-correcting programs

### Algorithm MultiplicationCheck

(Parameter: a natural number  $t$ )

Input: three natural numbers  $a$ ,  $b$  and  $c$  of size at most  $2^n$ .

Choose  $p$  uniformly at random among the  
first  $tn \log tn$  prime numbers.

Let  $d_1 = c \bmod p$ .

Let  $d_2 = ((a \bmod p)(b \bmod p)) \bmod p$ .

If  $d_1 = d_2$  then accept, else reject.

Algorithm MultiplicationCheck accepts an input  $a, b, c$

- ▶ with probability 1 in case  $ab = c$ ,
- ▶ with probability at most  $1/t$  in case  $ab \neq c$ .

The analysis is similar to the one of Algorithm Fingerprint.

## Checking and correcting: self-correcting programs

### Algorithm MultiplicationCorrection

Input: two natural numbers  $a$  and  $b$ , each represented by  $n$  bits.

Determine  $n$ -bit natural numbers  $r_1$  and  $r_2$   
by  $2n$  independent tosses of a fair coin.

Let  $c = (a + r_1)(b + r_2) - r_1(b + r_2) - r_2(a + r_1) + r_1r_2$  (\*)

Output:  $c$  (where the intended value of  $c$  is  $ab$ ).

The right-hand side of equation (\*) evaluates to the required value  $ab$  in case all arithmetical operations are correct.

Algorithm MultiplicationCorrection turns a multiplication procedure that *on certain rare inputs is always incorrect* into a multiplication procedure that *on any input is correct with high probability*.

## Checking and correcting: self-correcting programs

### Proposition

Assume that Algorithm MultiplicationCorrection is run on a processor where addition and subtraction are always correct, and multiplication is incorrect for a fraction of at most  $\varepsilon$  of all pairs of  $(n + 1)$ -bit numbers.

Then for any pair  $a$  and  $b$  of  $n$ -bit numbers, the probability that the algorithm fails to compute the product of  $a$  and  $b$  correctly is at most  $16\varepsilon$ .

### Proof.

It suffices to show that each of the four multiplications in (\*) is incorrect with probability at most  $4\varepsilon$ .

Observe that for all four multiplication both arguments are  $(n + 1)$ -bit natural numbers (where  $n$ -bit numbers are identified with appropriate  $(n + 1)$ -bit numbers).

## Checking and correcting: self-correcting programs

### Proof (continued).

We just consider the evaluation of the term  $r_1(b + r_2)$ , and omit the very similar consideration for the three remaining terms.

The numbers  $r_1$  and  $r_2$  are chosen uniformly and independently from all  $n$ -bit numbers, and the mapping  $x \mapsto b + x$  is injective.

Consequently, the pair  $(r_1, (b + r_2))$  is chosen uniformly from a set  $P$  of  $2^{2n}$  pairs of  $(n + 1)$ -bit numbers.

By assumption, multiplication is not correct

for at most an  $\varepsilon$ -fraction of all pairs of  $(n + 1)$ -bit numbers,  
that is, for at most  $\varepsilon 2^{2(n+1)} = 4\varepsilon 2^{2n}$  such pairs,

hence for at most an  $4\varepsilon$ -fraction of all pairs in  $P$ .  $\square$