

When File Synchronization Meets Number Theory

Antoine Amarilli, Fabrice Ben Hamouda, Florian Bourse,
David Naccache, and Pablo Rauzy

École normale supérieure, Département d’informatique
45, rue d’Ulm, F-75230, Paris Cedex 05, France.
`surname.name@ens.fr` (except for `fabrice.ben.hamouda@ens.fr`)

Abstract. In this work we [to be completed by David]

1 Introduction

This work revisits *set reconciliation*, a problem consisting in synchronizing two multisets while minimizing communication complexity. Set reconciliation has many practical applications, the most typical of which is certainly the incremental backup of information.

Set reconciliation has already efficient and elegant solutions. For instance, [1] presents a reconciliation protocol whose computational and communication complexities are linear in the number of differences in the reconciled multisets.

We refer the reader to [1,2,4] (to quote only a few typical references) for more on the problem’s history and the existing solutions.

This article proposes a new reconciliation protocol based on number theory. In terms of communication complexity, the proposed procedure is comparable to prior publications [1] (that anyhow reached optimality) but its main interest lies in the fact that the correctness of the proposed protocol can be reduced to the cryptographic security of specific signature padding schemes. This proof stems from the application of the approach described in [3].

In addition, the new protocol offers new parameter trade-offs and hence the possibility that specific implementations would offer *constant*-factor gains over alternative asymptotically-equivalent solutions.

2 A New Set Reconciliation Protocol

2.1 A Few Notations

We model the directory synchronization problem as follows: Oscar possesses an old version of a directory \mathfrak{D} that he wishes to update. Neil has the up-to-date version \mathfrak{D}' . The challenge faced by Oscar and Neil¹ is that of *exchanging as little data as possible* during the synchronization process. In practice \mathfrak{D} and \mathfrak{D}' usually differ both in their files and in their tree structure.

In tackling this problem this paper separates the “*what*” from the “*where*”: namely, we disregard the relative position of files in subdirectories and model directories as multisets of files. Let \mathfrak{F} and \mathfrak{F}' denote the multisets of files contained in \mathfrak{D} and \mathfrak{D}' . We denote $\mathfrak{F} = \{F_0, \dots, F_n\}$ and $\mathfrak{F}' = \{F'_0, \dots, F'_{n'}\}$.

¹ Oscar and Neil will respectively stand for *old* and *new*.

Let **Hash** denote a collision-resistant hash function² and let F be a file. Let $\text{NextPrime}(F)$ be the prime immediately larger than $\text{Hash}(F)$ and let u denote the size of NextPrime 's output in bits. Define the shorthand notations: $h_i = \text{NextPrime}(F_i)$ and $h'_i = \text{NextPrime}(F'_i)$.

TODO(amarilli): use the uniform nextprime (discussion of relative costs with respect to (1.) hashing costs and (2.) finding the next prime costs)

2.2 Description of the Basic Exchanges

Let t be the number of discrepancies between \mathfrak{F} and \mathfrak{F}' that we wish to spot, i.e.:

$$t = \#\mathfrak{F} + \#\mathfrak{F}' - 2\#(\mathfrak{F} \cap \mathfrak{F}')$$

We generate a prime p such that:

$$2^{2ut+1} \leq p < 2^{2ut+2} \quad (1)$$

Given \mathfrak{F} , Neil generates and sends to Oscar the redundancy:

$$c = \prod_{i=1}^n h_i \bmod p$$

Oscar computes:

$$c' = \prod_{i=1}^n h'_i \bmod p \quad \text{and} \quad s = \frac{c'}{c} \bmod p$$

Using [7] the integer s can be written as:

$$s = \frac{a}{b} \bmod p \quad \text{where the } G_i \text{ denote files and} \quad \begin{cases} a = \prod_{G_i \in \mathfrak{F}' \wedge G_i \notin \mathfrak{F}} \text{NextPrime}(G_i) \\ b = \prod_{G_i \notin \mathfrak{F}' \wedge G_i \in \mathfrak{F}} \text{NextPrime}(G_i) \end{cases}$$

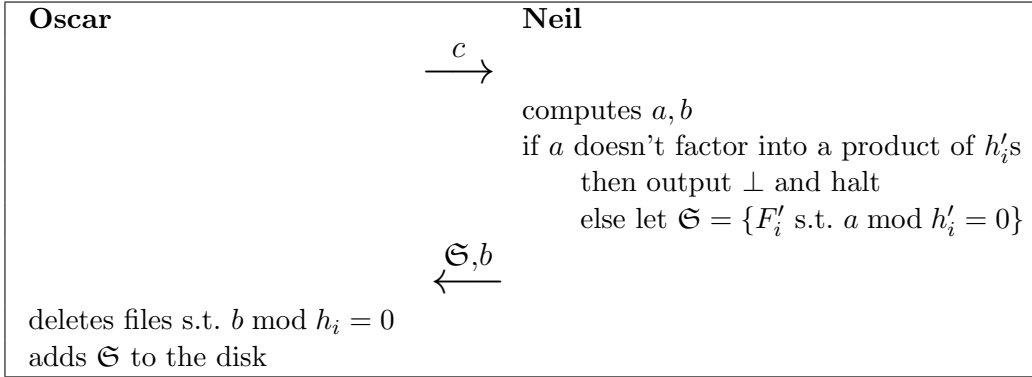
Note that since \mathfrak{F} and \mathfrak{F}' differ by at most t elements, a and b are strictly lesser than 2^{ut} . Theorem 1 (see [2]) guarantees a and b can be efficiently recovered from s (this problem is known as *Rational Number Reconstruction* [4,8]). This is done using Gauss' algorithm for finding the shortest vector in a bi-dimensional lattice [7].

Theorem 1. *Let $a, b \in \mathbb{Z}$ such that $-A \leq a \leq A$ and $0 < b \leq B$. Let $p > 2AB$ be a prime and $s = ab^{-1} \bmod p$. Then a, b can be recovered from A, B, s, p in polynomial time.*

Taking $A = B = 2^{ut} - 1$, (1) implies that $2AB < p$. Moreover, $0 \leq a \leq A$ and $0 < b \leq B$. Thus Oscar can recover a and b from s in polynomial time. By testing the divisibility of a and b by the h_i and the h'_i , Neil and Oscar can precisely and deterministically identify the discrepancies between \mathfrak{F} and \mathfrak{F}' and settle them.

Formally, this is done as follows:

² e.g. SHA-1



As we have just seen, the “output \perp and halt” protocol breakout should actually never occur as long as bounds on parameter sizes are respected. However, a file synchronization procedure that works *only* for a limited number of differences is not very useful in practice. Subsection explains how to use the protocol even when differences exceed t , the informational capacity of the modulus p .

2.3 Reducing Correctness to... The Security of RSA

RSA [?] is certainly the most famous public-key cryptosystem. Recall that to generate [?] system parameters, one defines a security parameter $k \in \mathbb{N}$ and an encoding function $\mu : \{0, 1\}^* \rightarrow \{0, 1\}^k$, selects two distinct $k/2$ -bit primes p, q and computes $N = pq$.

The public verification exponent is an integer $e \in \mathbb{Z}_{\phi(N)}^*$ and the corresponding secret signature exponent is $d = e^{-1} \bmod \phi(N)$.

To sign a message m , the signer \mathcal{S} returns the signature $y = \mu(m)^d \bmod N$.

To check the signature y , the verifier tests if $y^e \bmod N \stackrel{?}{=} \mu(m)$.

To reduce the correctness of the reconciliation protocol of section 2.2 to the security of RSA, we add two further restrictions to the RSA parameters: We additionally require that e is prime and $e > n + n'$. We call this RSA variant, *prime-exponent* RSA.

We also identify (redefine) two reconciliation protocol parameters with the following *prime-exponent* RSA parameters:

- Replace $h_i = \text{NextPrime}(F_i)$ by $h_i = \mu(F_i)$.
- Replace the modulus p used in section 2.2 by the RSA modulus N , i.e. redefine:

$$c = \prod_{i=1}^n \mu(F_i) \bmod N \quad \text{and} \quad c' = \prod_{i=1}^{n'} \mu(F'_i) \bmod N$$

If there exist an algorithm \mathcal{A} capable of computing a pair of multisets $(\mathfrak{F}, \mathfrak{F}')$ on which the reconciliation protocol is ineffective³, this implies that at the end of the protocol Oscar gets a multiset $\mathfrak{R} \neq \mathfrak{F}'$ for which:

³ Our definition of the term *ineffective* is: The protocol terminates considering that Oscar and Neil are in synchrony whilst they are not.

$$s = \frac{c}{c'} = 1 \bmod N \Rightarrow \prod_{R_i \in \mathfrak{R}} \mu(R_i) = \prod_{F_i \in \mathfrak{F}'} \mu(F_i) \bmod N$$

This is *exactly* the setting presented in [3], where multiset equality is ascertained by testing that:

$$\prod_{i=1}^n \mu(F_i) \stackrel{?}{=} \prod_{i=1}^{n'} \mu(F'_i) \bmod N$$

Hence, the security proof given in [3] implies the following theorem:

Theorem 2. *If there exists an algorithm \mathcal{A} such that:*

- *\mathcal{A} is capable of computing a pair of multisets $(\mathfrak{F}, \mathfrak{F}')$ on which the reconciliation protocol is ineffective,*
- *\mathcal{A} runs in time $t_2(k, \eta)$ and succeeds in finding $(\mathfrak{F}, \mathfrak{F}')$ with probability $\varepsilon_2(k, \eta)$, where $\eta = \max(\#\mathfrak{F}, \#\mathfrak{F}')$.*

then there exists a forger \mathcal{F} that finds a prime-exponent RSA forgery after $q_1(k, \eta) < 2\eta$ queries to \mathcal{S} and $t_1(k, \eta) = t_2(k, \eta) + \mathcal{O}(\eta^2) + \mathcal{O}(\eta k^2)$ computing time, with probability $\varepsilon_1(k, \eta) = \varepsilon_2(k, \eta)$.

2.4 The Case of Insufficient Information

To extend the protocol to an arbitrary t , Oscar and Neil agree on an infinite set of primes p_1, p_2, \dots . As long as the protocol fails, Neil will keep accumulating information about the difference between \mathfrak{F} and \mathfrak{F}' as shown in appendix A. Note that no information is lost and that the distilled modular knowledge about the difference adds-up until it reaches a threshold sufficient to reconcile \mathfrak{F} and \mathfrak{F}' .

To determine after each new p_i round if the synchronization is over, as the interaction starts Neil will send to Oscar $\text{Hash}(\mathfrak{F}')$. As long as Oscar's state does not match this target digest $\text{Hash}(\mathfrak{F}')$, Oscar will continue the interaction.

3 Efficiency Considerations

In this section we explore two strategies to reduce the size of p and hence improve transmission by *constant factors* (from an asymptotic communication complexity standpoint, nothing can be done as the protocol already transmits information proportional in size to the difference to settle).

3.1 Probabilistic Decoding: Reducing p

Generate a prime p about twice smaller than previously, namely:

$$2^{ut+w-1} < p \leq 2^{ut+w} \tag{2}$$

where $w \geq 1$ is some small integer (say $w = 50$). For large $\eta = \max(n, n')$ and t , the size of the new prime p used in this section will be approximately half the size of the prime p in

section 2.2. The new redundancy c is calculated as previously and is hence also approximately twice smaller. As previously, we have:

$$s = \frac{a}{b} \bmod p \quad \text{and} \quad \begin{cases} a = \prod_{G_i \in \mathfrak{F}' \wedge G_i \notin \mathfrak{F}} \text{NextPrime}(G_i) \\ b = \prod_{G_i \notin \mathfrak{F}' \wedge G_i \in \mathfrak{F}} \text{NextPrime}(G_i) \end{cases}$$

and since there are at most t differences, we must have:

$$ab \leq 2^{ut} \tag{3}$$

The difference with respect to section 2.2 is that we do not have a fixed bound for a and b anymore; equation (3) only provides a bound for the *product* ab . Therefore, we define a finite sequence of couples of integers:

$$(A_i, B_i) = (2^{wi}, \lfloor \frac{p-1}{2A_i} \rfloor) \quad \text{where } B_i > 1$$

For all $i > 0$ we have $2A_i B_i < p$. Moreover, from equations (2) and (3) there must exist at least one index i such that $0 \leq a \leq A_i$ and $0 < b \leq B_i$. Then using Theorem 1, given (A_i, B_i, p, s) one can recover a and b , and thus the difference between \mathfrak{F} and \mathfrak{F}' .

The problem is that (by opposition to the protocol of section 2.2) we have no guarantee that such an (a, b) is unique. Namely, we could (in theory) stumble by sheer bad luck over another (a', b') satisfying (3) for some index $i' \neq i$. We expect this to happen with negligible probability for large enough w , but this makes the modified protocol heuristic only.

To ascertain correctness upon termination, the parties can use the $\text{Hash}(\mathfrak{F}')$ protocol preamble mentioned in section 2.4.

3.2 The File Laundry: Reducing u

What happens if we shorten u in the basic protocol?

As foreseen by the birthday paradox, we should start seeing collisions. Let us analyze the statistics governing the appearance of collisions.

Consider Hash as a random function from $\{0, 1\}^*$ to $\{0, \dots, 2^u - 1\}$. Let X_i^1 be the random variable equal to 1 when the file F_i collides with another file, and equal to 0 otherwise. Clearly, we have $\Pr[X_i = 1] \leq \frac{\eta-1}{2^u}$. The average number of colliding files is hence:

$$\mathbb{E} \left[\sum_{i=0}^{\eta-1} X_i \right] \leq \sum_{i=0}^{\eta-1} \frac{\eta-1}{2^u} = \frac{\eta(\eta-1)}{2^u}.$$

For instance, for $\eta = 10^6$ files and 32-bit digests, the expected number of colliding files is less than 233.

That being said, note that a collision can only yield a *false positive* and never a *false negative*. In other words, while a collision may oblivate a difference⁴ a collision can never create a nonexistent difference *ex nihilo*.

⁴ e.g. make the parties blind to the difference between `index.htm` and `ieexplore.exe`.

Hence, it suffices to replace $\text{Hash}(F)$ by a chopped $\bar{h}_{k,u}(F) = \text{MAC}_k(F) \bmod 2^u$ to quickly filter-out file differences by repeating the protocol for $k = 1, 2, \dots$. At each iteration the parties will detect new files and new deletions, fix these and “launder” again the remaining multisets.

Indeed, the probability that a stubborn file persists colliding decreases exponentially with the number of iterations k , if MACs are random and independent from each other. Assume that η remains invariant between iterations. Define the following random variables:

$$X_i^\ell = \begin{cases} 1 & \text{if file } F_i \text{ collides with another file during iteration } \ell. \\ 0 & \text{otherwise.} \end{cases}$$

$$Y_i = \prod_{\ell=1}^k X_i^\ell = \begin{cases} 1 & \text{if file } F_i \text{ collides with another file for all the } k \text{ iterations.} \\ 0 & \text{otherwise.} \end{cases}$$

By independence, we have:

$$\Pr[Y_i = 1] = \prod_{\ell=1}^k \Pr[X_i^\ell = 1] = \Pr[X_i^1 = 1] \dots \Pr[X_i^k = 1] \leq \left(\frac{\eta-1}{2^u}\right)^k.$$

Therefore the average number of colliding files is:

$$\mathbb{E}\left[\sum_{i=0}^{\eta-1} Y_i\right] \leq \sum_{i=0}^{\eta-1} \left(\frac{\eta-1}{2^u}\right)^k = \eta \left(\frac{\eta-1}{2^u}\right)^k.$$

And the probability that at least one false positive will survive k rounds is:

$$\epsilon_k \leq \eta \left(\frac{\eta-1}{2^u}\right)^k$$

For the previously considered instance⁵ we get $\epsilon_2 \leq 5.43\%$ and $\epsilon_3 \leq 2 \cdot 10^{-3}\%$.

A more refined (but somewhat technical) analysis As mentioned previously, the parties can remove the files detected as different during the first iteration and only with common and colliding files. Now, the only collisions that can fool round k , are the collisions of a file-pairs (F_i, F_j) such that F_i and F_j have both already collided during *all the previous iterations*⁶. We call such collisions “stutters”. Define the random variable:

$$Z_i^\ell = \begin{cases} 1 & \text{if file } F_i \text{ stuttered during the } \ell \text{ first protocol iterations.} \\ 0 & \text{otherwise.} \end{cases}$$

⁵ $\eta = 10^6, u = 32$.

⁶ Note that the observation does not mandate that F_i and F_j who necessarily collided *which each other*. Indeed, we may witness during the first round $\bar{h}_{1,u}(F_1) = \bar{h}_{1,u}(F_2), \bar{h}_{1,u}(F_3) = \bar{h}_{1,u}(F_4)$ while during the second round $\bar{h}_{2,u}(F_1) = \bar{h}_{2,u}(F_4), \bar{h}_{2,u}(F_2) = \bar{h}_{2,u}(F_3)$.

Suppose that $\eta > 1$. Set $Z_i^0 = 1$ and write $p_\ell = \Pr \left[Z_i^{\ell-1} = 1 \text{ and } Z_j^{\ell-1} = 1 \right]$ for all ℓ and $i \neq j$. For $k \geq 1$, we have

$$\begin{aligned} \Pr \left[Z_i^k = 1 \right] &= \Pr \left[\exists j \neq i, X_{i,j}^k = 1, Z_i^{k-1} = 1 \text{ and } Z_j^{\ell-1} = 1 \right] \\ &\leq \sum_{j=0, j \neq i}^{\eta-1} \Pr \left[X_{i,j}^{k-1} = 1 \right] \Pr \left[Z_i^{k-1} = 1 \text{ and } Z_j^{k-1} = 1 \right] \\ &\leq \frac{\eta-1}{2^u} p_{k-1} \end{aligned}$$

Furthermore $p_0 = 1$ and

$$\begin{aligned} p_\ell &= \Pr \left[X_0^\ell = X_1^\ell, Z_0^\ell = 1 \text{ and } Z_1^\ell = 1 \right] + \Pr \left[X_0^\ell \neq X_1^\ell, Z_0^\ell = 1 \text{ and } Z_1^\ell = 1 \right] \\ &\leq \Pr \left[X_0^\ell = X_1^\ell, Z_0^{\ell-1} = 1 \text{ and } Z_1^{\ell-1} = 1 \right] \\ &\quad + \sum_{i \geq 2, j \geq 2} \Pr \left[X_{0,i}^\ell = 1, X_{1,j}^\ell = 1, Z_0^{\ell-1} = 1 \text{ and } Z_1^{\ell-1} = 1 \right] \\ &= \Pr \left[X_0^\ell = X_1^\ell \right] \Pr \left[Z_0^{\ell-1} = 1 \text{ and } Z_1^{\ell-1} = 1 \right] \\ &\quad + \sum_{i \geq 2, j \geq 2} \Pr \left[X_{0,i}^\ell = 1 \right] \Pr \left[X_{1,j}^\ell = 1 \right] \Pr \left[Z_0^{\ell-1} = 1 \text{ and } Z_1^{\ell-1} = 1 \right] \\ &\leq \frac{1}{2^u} p_{\ell-1} + \frac{(\eta-2)^2}{2^{2u}} p_{\ell-1} = p_{\ell-1} \left(\frac{1}{2^u} + \frac{(\eta-2)^2}{2^{2u}} \right) \end{aligned}$$

hence:

$$p_\ell \leq \left(\frac{1}{2^u} + \frac{(\eta-2)^2}{2^{2u}} \right)^\ell,$$

and

$$\Pr \left[Z_i^\ell = 1 \right] \leq \left(\frac{1}{2^u} + \frac{(\eta-2)^2}{2^{2u}} \right)^{k-1}$$

And finally, the survival probability of at least one false positive after k iterations satisfies:

$$\epsilon'_k \leq \frac{\eta(\eta-1)}{2^u} \left(\frac{1}{2^u} + \frac{(\eta-2)^2}{2^{2u}} \right)^{k-1}$$

For the $(\eta = 10^6, u = 32, k = 2)$ instance considered previously we get $\epsilon'_2 \leq 0.013\%$.

How to select u ? For a fixed k , ϵ'_k decreases as u grows. For a fixed u , ϵ'_k also decreases as k grows. Transmission, however, grows with both u (bigger digests) and k (more iterations). We denote for the sake of precision: $\epsilon'_k = \epsilon'_{k,u,\eta}$.

Fix η and note that the number of bits transmitted per iteration ($\simeq 3ut$), is proportional to u . This yields an expected transmission complexity $T_{u,\eta}$:

$$T_{u,\eta} \propto \tau = u \sum_{k=1}^{\infty} k \cdot \epsilon'_{k,u,\eta} = \frac{u\eta(\eta-1)}{2^u} \sum_{k=1}^{\infty} k \left(\frac{1}{2^u} + \frac{(\eta-2)^2}{2^{2u}} \right)^{k-1} = \frac{u\eta(\eta-1)2^{3u}}{(2^u - 2^{2u} + (\eta-2)^2)^2}$$

Dropping the proportionality factor $\eta(\eta-1)$, neglecting $2^u \ll 2^{2u}$ and approximating $(\eta-2) \simeq \eta$, we can optimize the function:

$$\phi_{\eta}(u) = \frac{u \cdot 2^{3u}}{(2^{2u} - \eta^2)^2}$$

It turns out that $\phi_{10^6}(u)$ reaches its optimum for $u = 19$.

Note: The previous bounds are rather crude, in particular:

- We consider u -bit prime digests while u -bit strings contain only about $2^u/u$ primes.
- In our probability calculations η can be replaced by the total number of differences t . It is reasonable to assume that in most *practical* settings $t \ll \eta$, but extreme instances where $t \sim \eta$ may exist as well.
- We used a fixed u in all rounds. Nothing forbids using a different u_k at each iteration⁷.

4 Theoretical Complexity and Algorithmic Improvements

In this section, we analyse the theoretical costs of our algorithms and propose some algorithmic improvements.

TODO(amarilli): we should compare the time complexity to that of the other paper, and, if we are better, insist on it. If we can combine this to “Practical set reconciliation”, we should.

4.1 Theoretical complexity

Let $M(k)$ be the time required to multiply two numbers of k bits. We suppose $M(k+k') \geq M(k) + M(k')$, for any k, k' . We know that the division and the modular reduction of two numbers of k bits modulo a number of k bits costs $\tilde{O}(M(k))$ [1]. Furthermore, using naive algorithms, $M(k) = O(k^2)$, but using fast algorithms such as FFT [5], $M(k) = \tilde{O}(k)$. We note that the FFT multiplication is faster than the other methods (naive or Karatsuba) for number of about $10^4 \cdot 64$ bits (from gmp sources – if you find any better sources, it would be interesting...). And using such big numbers, the division and the modulo reduction algorithms used in gmp are also the ones with complexity $\tilde{O}(M(k))$.

Since p has $2ut$ bits, here are the costs:

1. (Neil) computation of the redundancy $c = \prod_{i=1}^n h_i \bmod p$, cost: $O(nM(ut))$, $\tilde{O}(nut)$ with FFT
2. (Oscar) computation of the redundancy $c' = \prod_{i=1}^n h_i \bmod p$, cost: $O(nM(ut))$, $\tilde{O}(nut)$ with FFT

⁷ ...or even fine-tuning the u_k adaptively, as a function of the laundry's effect on the progressively reconciliated multisets.

3. (Oscar) computation of $s = c'/c \bmod p$, cost: $M(ut)$, $\tilde{O}(ut)$ with FFT
4. (Oscar) computation of the two ut -bits number a and b , such that $s = a/b \bmod p$, cost: $\tilde{O}(M(ut))$, using a new technique of Wang and Pan in [4] and [8]; however using naive extended gcd, it costs $\tilde{O}((ut)^2)$. @fbenhamo TODO However I do not know any software where it is implemented, nor the actual speed in practice, neither if this can be adapted for the polynomial case (this can be an advantage over the polynomial method for set reconciliation – but I think this is not the case, unfortunately, I have not access to interesting articles about polynomial rational reconstruction - but see p.139 of <http://algo.inria.fr/chyzak/mpri/poly-20120112.pdf>).
5. (Oscar) factorization of a , i.e., n modulo reductions of a by a h_i , cost: $\tilde{O}(nM(ut))$, $\tilde{O}(nut)$ with FFT
6. (Oscar) factorization of b , i.e., n modulo reductions of b by a h_i , cost: $\tilde{O}(nM(ut))$, $\tilde{O}(nut)$ with FFT

4.2 Improvements

It is possible to improve the complexity of the computation of the redundancy and the factorization to $\tilde{O}(n/tM(ut))$, $\tilde{O}(nu)$ with FFT [5]. To simplify the explanations, let us suppose t is a power of 2: $t = 2^\tau$, and t divides n .

The idea is the following: we group h_i by group of t elements and we compute the product of each of these groups (without modulo)

$$H_j = \prod_{i=jt}^{jt+t-1} h_i.$$

Each of these products can be computed in $\tilde{O}(M(ut))$ using a standard method of product tree, depicted in Algorithm 1 (for $j = 0$) and in Figure 1. And all these n/t products can be computed in $\tilde{O}(n/tM(ut))$. Then, one can compute c by multiplying these products H_j together, modulo p , which costs $\tilde{O}(n/tM(ut))$.

Algorithm 1 Product tree algorithm

Require: a table h such that $h[i] = h_i$

Ensure: $\pi = \pi_1 = \prod_{i=0}^{t-1} h_i$, and $\pi[i] = \pi_i$ for $i \in \{1, \dots, 2t-1\}$ as in Figure 1

```

1:  $\pi \leftarrow$  array of size  $t$ 
2: function PRODTREE( $i, \text{start}, \text{end}$ )
3:   if  $\text{start} = \text{end}$  then
4:     return 1
5:   else if  $\text{start} + 1 = \text{end}$  then
6:     return  $h[\text{start}]$ 
7:   else
8:      $\text{mid} \leftarrow \lfloor (\text{start} + \text{end})/2 \rfloor$ 
9:      $\pi[i] \leftarrow$  PRODTREE( $2 \times i, \text{start}, \text{mid}$ )
10:     $\pi[i + 1] \leftarrow$  PRODTREE( $2 \times i + 1, \text{start}, \text{mid}$ )
11:    return  $\times$  PRODTREE( $\text{mid}, \text{end}$ )
12:  $\pi[1] \leftarrow$  PRODTREE( $1, 0, t$ )
```

The same technique applies for the factorization, but this time, we have to be a little more careful. After computing the tree product, we can compute the residues of a (or b) modulo H_j , then we can compute the residues of these new elements modulo the two children of H_j in the product tree ($\prod_{i=jt}^{jt+t/2-1} h_i$ and $\prod_{i=jt}^{jt+t/2-1} h_i$), and then compute the residues of these two new values modulo the children of the previous children, and so on. Intuitively, we go down the product tree doing modulo reduction. At the end (i.e., at the leaves), we obtain the residues of a modulo each of the h_i . This algorithm is depicted in Algorithm 2 and in Figure 2 (for $j = 1$). The complexity of the algorithm is $\tilde{O}(M(ut))$, for each j . So the total complexity is $\tilde{O}(n/t\tilde{O}(M(ut)))$.

Algorithm 2 Division using product tree

Require: a an integer, π the product tree from Algorithm 1
Ensure: $A_i = A[i] = a \bmod \pi_i$ for $i \in \{1, \dots, 2t - 1\}$, computed as in Figure 2

```

1:  $A \leftarrow$  array of size  $t$ 
2: function MODTREE( $i$ )
3:   if  $i < 2t$  then
4:      $A[i] \leftarrow A[\lfloor i/2 \rfloor] \bmod \pi[i]$ 
5:     MODTREE( $2 \times i$ )
6:     MODTREE( $2 \times i + 1$ )
7:  $A[1] \leftarrow a \bmod \pi[1]$ 
8: MODTREE(2)
9: MODTREE(3)
```

5 Optimizing Parameters

The proposed process lends itself to a final fine-tuning. We list here some of the proposed research directions that could be investigated to that end:

5.1 Using a Smooth p

Comme explique dans un ancien email, je pense que l'on devrait utiliser un produit de petits nombres premiers au lieu d'un grand nombre premier p . Des l'instant que ces petits nombres premiers sont plus grands que les hashes, cela fonctionne. L'intérêt est que l'on peut travailler modulo ces "petits nombres premiers" avec le CRT. Et en plus, la generation de ce modulo p (pas premier) est beaucoup plus rapide.

Faster hashing into the primes using $h - h \bmod \pi + k\pi$ where π is a product of small primes

6 Implementation

To illustrate the concept, the authors has coded and evaluated the proof of concept described in this section.

The executable and source codes of the program, called `btrsyc`, can be downloaded from: <https://github.com/RobinMorisset/Btrsyc>.

The synchronisation is unidirectional (clearer). The program consist in two subprograms: a bash script and a python script:

6.1 The Bash Script

A bash script runs a python script (describe below) on the two computers to be synchronized. If the computer is not the one running the bash script, the python script is executed through ssh. The bash scripts also creates two pipes: one from Neil stdin to Oscar stdout and one from Oscar stdin to Neil stdout. Data exchanged during the protocol transits via these two pipes.

6.2 The Python Script

The python script uses gmp which implements all the number theory operations required by Oscar and Neil, and does the actual synchronization. This script works in two phases:

Finding Different Files

1. Compute the hashes of all files concatenated with their paths, type (folder/file), and permissions (not supported yet).
2. Implement the protocol proposed in Section ?? [add here a reference to the appropriate section in the paper] with input data coming from stdin and output data going to stdout.

More precisely:

- Oscar sends it product of hashes modulo a first prime number p_1 .
- Neil receives the product, divides by its own product of hashes, reconstructs the fraction modulo p_1 [can we elaborate more on what happens here? which functions in GMP are used to do the reconstruction?] and checks if he can factor the denominator using his hashes base. If he can, he stops and sends the numerator and the list of tuples (path, type, hash of content of the file) corresponding to the denominator's factors. Otherwise he sends "None" [is this the ASCII string "None"? if not what does he send precisely?].
- If Neil sent "None", Oscar computes the product of hashes modulo another prime p_2 , sends it... CRT mechanism... [can we elaborate more on what happens here? which functions in GMP are used to do the CRT?]
- If Neil sent the numerator and a list of tuples, then Oscar factors the numerator over his own hash values. Now each party (Neil, Oscar) knows precisely the list of files (path + type + hash of content) that differs from the over party.

[please structure the following:]

2. synchronize all the stuff [this is not an expression we can use in a paper...]. This part is not completely optimized.

We just remove all folders Oscar should not have and create new folders.

Then we remove all files Oscar should not have and synchronize using rsync the last files.

We could check for move (since we have the list of hash of contents of files) and do moves locally.

We can even try to detect moves of complete subtrees...

Capture the following:

6.3 Move Resolution Algorithm

To reproduce the structure of Oscar on Neil, we have a list of file moves to apply. Sadly, it is not straightforward to apply the moves, because, if we take a file to move, its destination might be blocked, either because a file already exists (we want to move a to b , b already exists), or because a folder cannot be created (we want to move a to b/c , b already exists but is a file and not a folder). Note that for a move operation $a \rightarrow b$, there is at most one file blocking the location b : we will call it the *blocker*.

If the blocker is not present on Oscar, then we can just delete. However, if it exists, then we might need to move it somewhere else before we solve the move we are interested in. This move itself might have a blocker, and so on. It seems that we just need to continue until we reach a move which has no blocker or where the blocker can be deleted, but we can get caught in a cycle: if we must move a to b , b to c and c to a , then we will not be able to perform anything without using a temporary location.

How can we perform the moves? A simple way would be to move each file to a unique temporary location and then rearrange files to our liking: however, this performs many unnecessary moves and will result in problems if the program is interrupted. We can do something more clever so by performing a decomposition in strongly connected components of the *move graph* (with one vertex per file and one edge per move operation going from the file to its blocker or to its destination if no blocker exists). The computation of the SCC decomposition is simplified by the observation that because two files being moved to the same destination must be equal, we can only keep one arbitrary in-edge per node, and look at the graph pruned in this fashion: its nodes have in-degree at most one, so the strongly connected components are either single nodes or cycles. Once the SCC decomposition is known, the moves can be applied by applying each SCC in a bottom-up fashion, an SCC's moves being solved either trivially (for single files) or using one intermediate location (for cycles).

The detailed algorithm is implemented as two mutually recursive functions and presented as Algorithm 3.

An optimization implemented by **btrfsync** over the algorithm described here is to move files instead of copying them and then removing the original file, because moves are faster than copies on most filesystems as they don't need to copy the file contents.

6.4 Experimental Comparison to rsync

To demonstrate the benefits of our approach, we compared our **btrfsync** implementation to the standard **rsync** on the following datasets:

- synthetic** A directory containing 1000 very small files containing the numbers from 1 to 1000.
- synthetic.shuffled** The result of applying a few operations to **synthetic**: 10 files were deleted, 10 files were renamed, and the contents of 10 files was changed.
- source** A snapshot of the **btrfsync** source tree.
- source.moved** The result of renaming a big folder (several hundred of kilobytes) in **source**.
- firefox-13.0** The source archive of Mozilla Firefox 13.0.
- firefox-13.0.1** The source archive of Mozilla Firefox 13.0.1.
- empty** An empty folder.

Algorithm 3 Perform moves

Require: D is a dictionary where $D[f]$ denotes the intended destinations of f

```

1:  $M \leftarrow []$ 
2:  $T \leftarrow []$ 
3: for  $f$  in  $D$ 's keys do
4:    $M[f] \leftarrow \text{not\_done}$ 
5: function UNBLOCK_COPY( $f, t$ )
6:   if  $t$  is blocked by some  $b$  then
7:     if  $b$  is not in  $D$ 's keys then
8:        $\text{unlink}(b)$  ▷ We don't need  $b$ 
9:     else
10:       $\text{RESOLVE}(b)$  ▷ Take care of  $b$  and make it go away
11:   if  $T[f]$  was set then
12:      $f \leftarrow T[f]$ 
13:    $\text{copy}(f, d)$ 
14: function RESOLVE( $f$ )
15:   if  $M[f] = \text{done}$  then
16:     return ▷ Already managed by another in-edge
17:   if  $M[f] = \text{doing}$  then
18:      $T[f] \leftarrow \text{mktemp}()$ 
19:      $\text{move}(f, T[f])$ 
20:      $M[f] \leftarrow \text{done}$ 
21:     return ▷ We found a loop, moved  $f$  out of the way
22:    $M[f] \leftarrow \text{doing}$ 
23:   for  $d \in D[f]$  do
24:     if  $d \neq f$  then
25:        $\text{unblock\_copy}(f, d)$  ▷ Perform all the moves
26:   if  $f \notin D[f]$  and  $T[f]$  was not set then
27:      $\text{unlink}(f)$ 
28:   if  $T[f]$  was set then
29:      $\text{unlink}(T[f])$ 
30: for  $f$  in  $D$ 's keys do
31:    $\text{RESOLVE}(f)$ 

```

We performed the measurements with **rsync** version 3.0.9 (used both as the standalone **rsync** and for the underlying call to **rsync** in **btrsycn**). The standalone **rsync** was given the `--delete` flag to delete existing files on Oscar which do not exist on Neil. We passed the `-I` flag to ensure that **rsync** did not cheat by looking at the file modification times (which **btrsycn** does not do), and `--chmod="a=rx,u+w"` as an attempt to disable the transfer of file permissions (which **btrsycn** does not transfer). Although these settings ensure that **rsync** does not need to transfer the permissions, verbose logging suggests that it transfers them anyway, so **rsync** must lose a few bytes per file as compared to **btrsycn** for this reason.

Bandwidth accounting was performed with the `-v` flag for **rsync** invocations (which report the number of sent and received bytes), and by counting the amount of data transmitted for **btrsycn**'s own negotiations. The experiment was performed between two remote hosts on a high-speed link. The time measurements account both for CPU time and for transfer time and are just given as a general indication.

The results are given in table 3. The timing results show that **btrsycn** is slower than **rsync**, especially when the number of files is high (i.e., the synthetic datasets). The bandwidth results, however, are more satisfactory. It is true that the trivial datasets where either Oscar or Neil have no data, **rsync** outperforms **btrsycn**: this is especially clear in the case where Neil has no data: **rsync** immediately notices that there is nothing to transfer, whereas **btrsycn** transfers information to determine the symmetric difference. On the non-trivial datasets, however, **btrsycn** outperforms **rsync**. This is the case on the synthetic datasets, where **btrsycn** does not have to transfer information about all the files which were not modified, and even more so on the case where there are no modifications at all. On the Firefox source code dataset, **btrsycn** saves a very small amount of bandwidth, presumably because of unmodified files. For the **btrsycn** source code dataset, we notice that **btrsycn**, unlike **rsync**, was able to detect the move and to avoid retransferring the moved folder.

7 Haskell Implementation

(Antoine: je range ça dans sa propre section, mais j'imagine qu'on va le couper ou juste dire que ça existe.)

7.1 Program Structure

A proof-of-concept called Btrsycn has been implemented in Haskell and is available at <https://github.com/RobinMo>

It is intended to work as a drop-in replacement of **rsync** for directories, taking as arguments two (possibly remote) directories. It launches instances of itself on each of these machines (by `ssh`), playing respectively Neil and Oscar's roles.

Communication between Neil and Oscar is handled by the original instance, that links each agent standard output to the standard input of the other.

Neil does almost all computations, while Oscar send him the needed informations and run the effective transfer of files when the computations are done. Btrsycn uses **rsync** to synchronize single files, because it's algorithm to detect changes in a files is very good.

7.2 Time Measurements

Because of difficulties in linking with the GMP library the code is significantly slower than it could be (especially in the computation of the primes from the hashes).

TODO: benchmarks with time + bandwidth (our only benefit ..)

8 Conclusion and Further Improvements

In this work we [to be completed by David]

Mention that the determination of the optimal u is an interesting open question

9 Acknowledgment

The authors acknowledge Guillaín Potron for his early involvement in this research work.

todo: Fix euclidean to Euclidean in reference 5.

todo: Merge two reference files rsynch and wagner.

References

1. Burnikel, C., Ziegler, J., Im Stadtwald, D., et al.: Fast recursive division (1998)
2. Fouque, P.A., Stern, J., Wackers, J.G.: Cryptocomputing with rationals. In: Blaze, M. (ed.) Financial Cryptography. Lecture Notes in Computer Science, vol. 2357, pp. 136–146. Springer (2002)
3. Hofmann, M., Pierce, B.C., Wagner, D.: Edit lenses. In: Field, J., Hicks, M. (eds.) POPL. pp. 495–508. ACM (2012)
4. Pan, V., Wang, X.: On rational number reconstruction and approximation. SIAM Journal on Computing 33, 502 (2004)
5. Schönhage, A., Strassen, V.: Schnelle multiplikation grosser zahlen. Computing 7(3), 281–292 (1971)
6. Tridgell, A.: Efficient algorithms for sorting and synchronization. Ph.D. thesis, PhD thesis, The Australian National University (1999)
7. Vallée, B.: Gauss’ algorithm revisited. J. Algorithms 12(4), 556–572 (1991)
8. Wang, X., Pan, V.: Acceleration of euclidean algorithm and rational number reconstruction. SIAM Journal on Computing 32(2), 548 (2003)

References

1. Y. Minsky, A. Trachtenberg, *Scalable Set Reconciliation*, 40th Annual Allerton Conference on Communications, Control and Computing, Monticello, IL, October 2002. A full version entitled *Practical Set Reconciliation* can be downloaded from <http://ipsit.bu.edu/documents/BUTR2002-01.ps>
2. Y. Minsky, A. Trachtenberg, R. Zippel, (2003). *Set reconciliation with nearly optimal communication complexity*. IEEE Transactions on Information Theory, 49(9), pp. 2213–2218.
3. Julien Cathalo, David Naccache, Jean-Jacques Quisquater, Comparing with RSA, Cryptography and Coding, Lecture Notes in Computer Science Volume 5921, 2009, pp 326-335, Cryptography and Coding, 12th IMA International Conference, Cryptography and Coding 2009, Cirencester, UK, December 15-17, 2009. Proceedings. Springer-Verlag Berlin Heidelberg
4. D. Eppstein, M. Goodrich, F. Uyeda, G. Varghese What’s the difference?: efficient set reconciliation without prior context ACM SIGCOMM Computer Communication Review - SIGCOMM ’11, 41(4), pp. 218-229, 2011.

A Extended Protocol

First phase during which Neil amasses modular information on the difference	
Oscar	Neil start protocol with p_1 $\xrightarrow{c_1}$ computes a, b using p_1 if a factors properly then go to Final Phase else perform protocol with p_2 $\xrightarrow{c_2}$ computes $c \bmod p_1 p_2 = \text{CRT}_{p_1, p_2}(c_1, c_2)$ computes a, b using $p_1 p_2$ if a factors properly then go to Final Phase else perform protocol with p_3 $\xrightarrow{c_3}$ computes $c \bmod p_1 p_2 p_3 = \text{CRT}_{p_1, p_2, p_3}(c_1, c_2, c_3)$ computes a, b using $p_1 p_2 p_3$ if a factors properly then go to Final Phase else perform protocol with p_4 \vdots
Final Phase	
$\xleftarrow{\mathfrak{S}, b}$ deletes files s.t. $b \bmod h_i = 0$ adds \mathfrak{S} to the disk	Let $\mathfrak{S} = \{F'_i \text{ s.t. } a \bmod h'_i = 0\}$

Note that the parties do not need to store the p_i 's in full. Indeed, the p_i could be subsequent primes sharing their most significant bits. This reduces storage per prime to a small corrected additive constant $\cong \ln(p_i) \cong \ln(2^{2tu+2}) \cong 1.39(tu+1)$ whose storage requires essentially $\log_2(tu)$ bits.

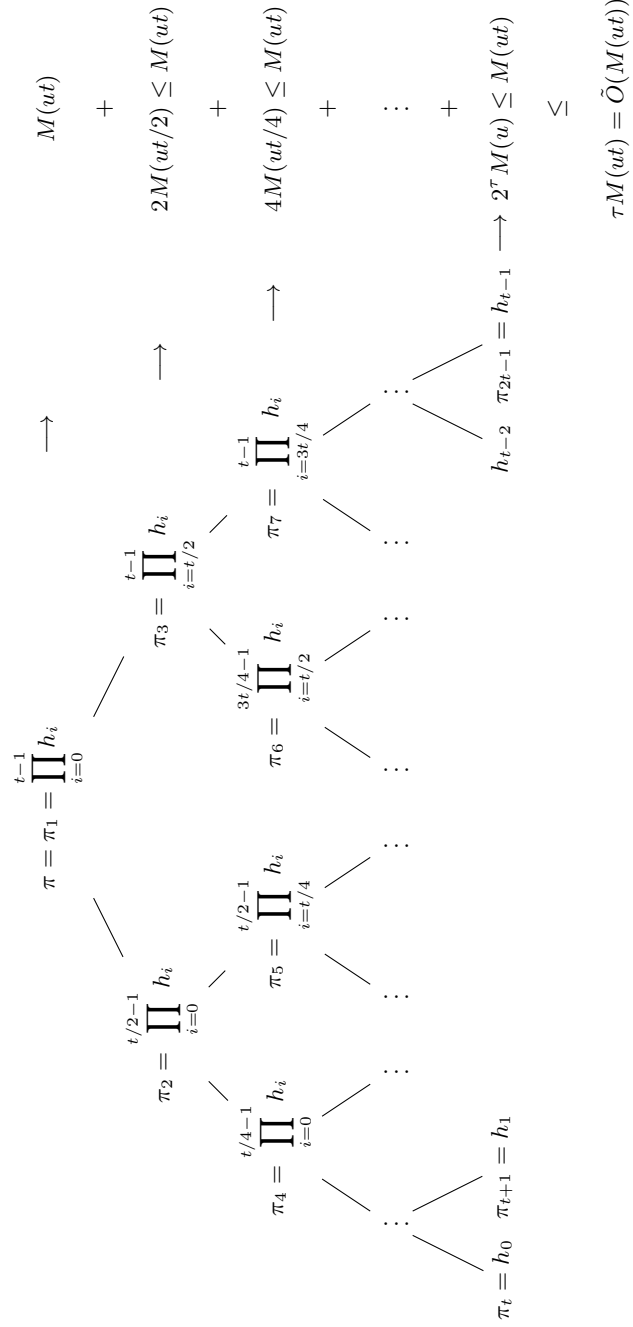


Fig. 1. Product tree

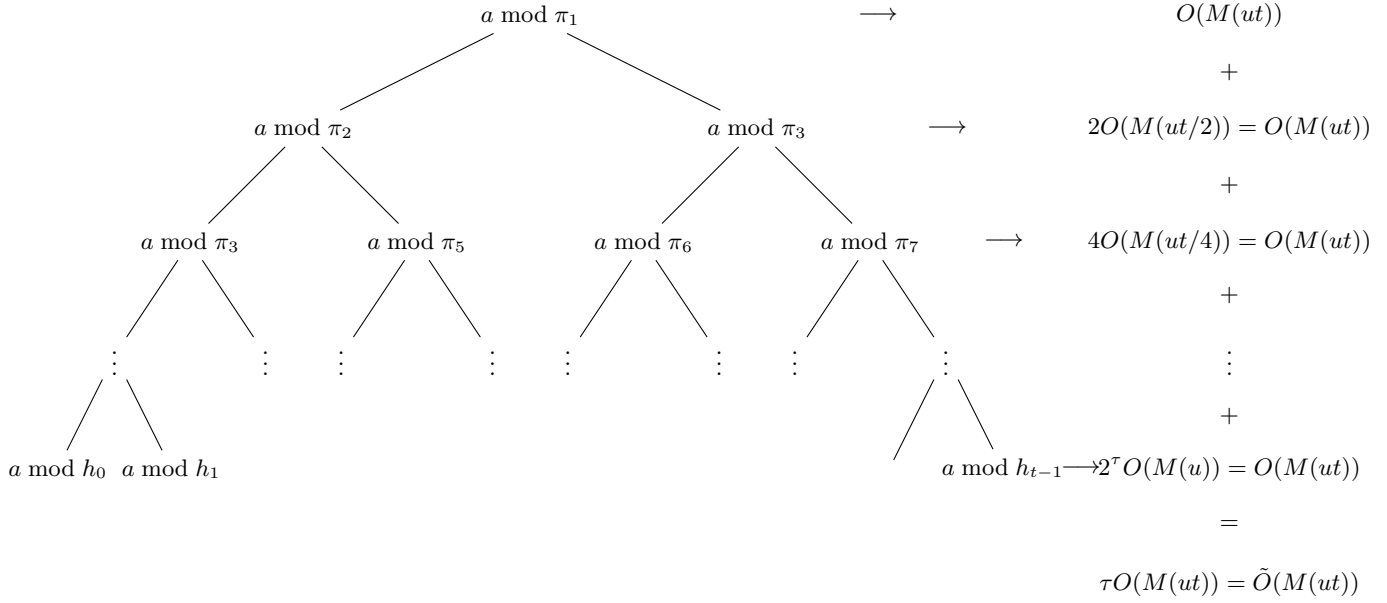


Fig. 2. Division from product tree

Datasets		Bandwidth (bytes)						Time (s)	
Neil	Oscar	TX _r	RX _r	TX _b	RX _b	abs	rel	t _r	t _b
source	empty	1613	778353	1846	788357	10237	+2 %	0.2	7.7
empty	source	11	29	12436	6120	18516	+46305 %	0.1	5.5
empty	empty	11	29	19	28	7	+32 %	0.1	0.3
synthetic	synthetic_shuffled	24891	51019	3638	4147	-68125	-57 %	0.2	26.8
synthetic_shuffled	synthetic	24701	50625	3443	3477	-68406	-58 %	0.2	26.6
synthetic	synthetic	25011	50918	327	28	-75574	-67 %	0.1	25.7
firefox-13.0.1	firefox-13.0	90598	28003573	80895	27995969	-17307	+0 %	2.6	4.2
source_moved	source	2456	694003	1603	1974	-692882	-99 %	0.2	2.5

Fig. 3. Experimental results. The two first columns indicate the datasets, synchronization is performed *from* Neil *to* Oscar. RX and TX are received and sent byte counts, r and b are **rsync** and **btrsync**, we also provide the absolute difference in exchanged data (positive when **btrsync** transfers more data than **rsync**) and the relative amount of data sent by **btrsync** compared to **rsync** (over 100% when **btrsync** transfers more data than **rsync**). The last two columns show timing results.