From Rational Number Reconstruction to Set Reconciliation and File Synchronization

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

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

Abstract. This work revisits *set reconciliation*, the problem of synchronizing two multisets of fixed-size values while minimizing transmission complexity. We propose a new number-theoretic reconciliation protocol called *Divide and Factor* (D&F) that achieves optimal asymptotic transmission complexity – as do previously known alternative algorithms. We analyze the computational complexities of various D&F variants, study the problem of synchronizing sets of variable-size files using hash functions and apply D&F to synchronize file hierarchies taking file locations into account.

We describe `btrsync`, our open-source D&F implementation, and benchmark it against the popular software `rsync`. It appears that `btrsync` transmits much less data than `rsync`, at the expense of a relatively modest computational overhead.

1 Introduction

File synchronization is the important practical problem of retrieving a file hierarchy from a remote host given an outdated version of the retrieved files. In many cases, the bottleneck is network bandwidth. Hence, transmission must be optimized using the information given by the outdated files to the fullest possible extent. Popular file synchronization programs such as `rsync` use rolling checksums to skip remote file parts matching local file parts; however, such programs are usually unable to use the outdated files in more subtle ways, *e.g.*, detect that information is already present on the local machine but at a different location or under a different name.

File synchronization is closely linked to the theoretical *Set Reconciliation Problem*: given two sets of fixed-size data items on different machines, determine the sets' symmetric difference while minimizing transmission complexity. The size of the symmetric difference (*i.e.*, the difference's cardinality times the elements' size) is a clear information-theoretic lower bound on the quantity of information to transfer, and several known algorithms already achieve this bound [10].

This paper considers set reconciliation and file synchronization from both a theoretical and practical perspective:

- Section 2 introduces *Divide and Factor* (D&F), a new number-theoretic set reconciliation algorithm. D&F represents the items to synchronize as prime numbers, accumulates information during a series of rounds and computes the sets' difference using Chinese remaindering and rational number reconstruction.
- Section 3 shows that D&F's transmission complexity is linear in the size of the symmetric difference of the multisets to reconcile.
- Section 4 extends D&F to perform *file reconciliation*, *i.e.*, reconcile sets of variable-size files. We show how to choose the hash functions to optimally trade transmission for success probability. Several elements in this analysis are generic and apply to all set reconciliation algorithms.
- Section 5 studies D&F's time complexity and presents constant-factor trade-offs between transmission and computation.
- Section 6 compares D&F with existing set reconciliation algorithms.
- Section 7 extends D&F to *file synchronization*, taking into account file locations and dealing intelligently (*i.e.*, in-place) with file moves. We describe an algorithm applying a sequence of file moves while avoiding the excessive use of temporary files.

– Section 8 presents `btrsync`, our D&F implementation, and benchmarks it against `rsync`. Experiments reveal that `btrsync` requires more computation than `rsync` but transmits less data in most cases.

2 Divide and Factor Set Reconciliation

This section introduces *Divide and Factor*. After introducing the problem and notations, we present a *basic* D&F version assuming that the number of differences between the multisets to reconcile is bounded by some constant t known to the parties. We then extend this basic protocol to a *complete* algorithm dealing with any number of differences.

2.1 Problem Definition and Notations

Oscar possesses an old version of a multiset $\mathcal{H} = \{h_1, \ldots, h_n\}$ that he wishes to update. Neil has a newer, up-to-date multiset $\mathcal{H}' = \{h'_1, \ldots, h'_{n'}\}$. The h_i, h'_i are u-bit primes. Note that Neil does not need to learn $\mathcal{H}$[1].

Let $\mathcal{D} = \mathcal{H} \setminus \mathcal{H}'$ be the multiset of values owned by Neil but not by Oscar, with adequate multiplicities. Likewise, let $\mathcal{D}' = \mathcal{H}' \setminus \mathcal{H}$ be the values owned by Oscar and not by Neil.

Let $T = \#\mathcal{D} + \#\mathcal{D}' = \#(\mathcal{D} \Delta \mathcal{D}')$ be the number of differences between $\mathcal{H}$ and $\mathcal{H}'$, where $\mathcal{D}\Delta\mathcal{D} = (\mathcal{D} \setminus \mathcal{D}') \cup (\mathcal{D}' \setminus \mathcal{D})$.

2.2 Basic Protocol with Bounded T

Assume that $T \leq t$ for some fixed t known by Neil and Oscar. The initial phases of the protocol are as follows:

– Generate a prime p such that $2^{2ut} \leq p < 2^{2ut+1}$.
– Oscar computes the *redundancy* $c = \prod_{i=1}^{n} h_i \bmod p$ and sends it to Neil.
– Neil computes $c' = \prod_{i=1}^{n} \bmod p$ and $s = \frac{c'}{c} \bmod p$.

Because $T \leq t$, $\mathcal{H}$ and $\mathcal{H}'$ differ by at most t elements and s can be written as follows:

$$s = \frac{a}{b} \bmod p \text{ where } \begin{cases} a = \prod_{h'_i \in \mathcal{H}' \setminus \mathcal{H}} h'_i \leq 2^{ut} - 1 \\ b = \prod_{h_i \in \mathcal{H} \setminus \mathcal{H}'} h_i \leq 2^{ut} - 1 \end{cases}.$$

The problem of efficiently recovering a and b from s is called *Rational Number Reconstruction* (RNR) [11,15]. The following theorem (*cf.* Theorem 1 of [7]) guarantees that RNR can be solved efficiently in the present setting:

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

Taking $A = B = 2^{ut} - 1$, we have $AB < p$, since $2^{2ut} \leq p < 2^{2ut+1}$. Moreover, $0 \leq a \leq A$ and $0 < b \leq B$. Thus Oscar can recover a and b from s in polynomial time *e.g.*, using Gauß's algorithm for finding the shortest vector in a bi-dimensional lattice [14].

Oscar and Neil can then test, respectively, the divisibility of a and b by elements of the sets $\mathcal{H}$ and $\mathcal{H}'$ to identify the differences between $\mathcal{H}$ and $\mathcal{H}'$ and settle them[2]. This basic protocol is depicted in Figure 1.

[1] The protocol can be easily transformed to do so without changing asymptotic transmission complexities.
[2] Actually, this only works if $\mathcal{H}$ and $\mathcal{H}'$ are sets. In the case of multisets, if the multiplicity of h'_i in $\mathcal{H}'$ is j', then we would need to check the divisibility of b by $h_i, h_i^2, \ldots, h_i^{j'}$. For the sake of clarity we will assume that $\mathcal{H}$ and $\mathcal{H}'$ are sets. Adaptation to the general case is straightforward.

Oscar		Neil
$c \leftarrow \prod_{i=1}^{n} h_i \bmod p$	$\xrightarrow{c}$	$c' \leftarrow \prod_{i=1}^{n'} h'_i \bmod p$ $s \leftarrow c'/c \bmod p$ reconstruct a, b from s
$\mathcal{D} \leftarrow \{h_i \in \mathcal{H} \mid b \bmod h_i = 0\}$	$\xleftarrow{\mathcal{D}', b}$	$\mathcal{D}' \leftarrow \{h'_i \in \mathcal{H}' \mid a \bmod h'_i = 0\}$

Fig. 1. Basic D&F protocol (assuming that $T \leq t$).

2.3 Full Protocol with Unbounded T

In practice, we cannot assume that we have an upper bound t on the number of differences T. This section extends the protocol to any T. We do this in two steps. We first show that we can slightly change the protocol to detect whether a choice of t was large enough for a successful reconciliation (which is guaranteed to be true if t was $\geq T$). We then construct a protocol that works with any T.

Detecting Reconciliation Failures. If $t < T$, with high probability, a will not factor completely over the set of primes $\mathcal{H}'$[3]. We will (improperly) consider that in such a case a is a random (tu)-bit number. For each i, the probability that h'_i divides a is at most $1/2^u$. The probability that $\prod h'_i \mod a = 0$ is roughly the probability that exactly t h'_i's amongst n' divide a, *i.e.*, $\binom{n'}{t} 2^{-ut}(1-2^{-u})^{n'-t} \leq n2^{-ut}$ which is very small if $2^u \gg n'$, a condition that we assume hereafter.

Thus, Neil can check very quickly that $\prod h'_i \mod a = 0$ without sending any data to Oscar. We call $\perp_1$ the event where this test failed (which implies that reconciliation failed), and $\perp_2$ the event where this test succeeds but reconciliation failed (which is very unlikely according to the previous discussion).

To handle $\perp_2$, we will use a collision-resistant hash function `Hash`, such as SHA: Before any exchanges take place, Neil will send to Oscar $H = \texttt{Hash}(\mathcal{H}')$. After computing $\mathcal{D}$, Oscar will compute a candidate $\mathcal{H}'$ from $\{\mathcal{H}, \mathcal{D}', \mathcal{D}\}$ and check that this candidate $\mathcal{H}'$ hashes into H. As `Hash` is collision-resistant, we can detect event $\perp_2$ in this fashion.

Complete D&F Protocol. To extend the protocol to an arbitrary T, assume that Oscar and Neil agree on an infinite set of primes $p_1, p_2, \ldots$ As long as $\perp_1$ or $\perp_2$ occurs, Neil and Oscar will repeat the protocol with a new p_ℓ to learn more information on $\mathcal{H}'$. Oscar will keep accumulating information about the difference between $\mathcal{H}$ and $\mathcal{H}'$ during these protocol runs (called *rounds*).

Formally, assume that $2^{2ut_k} \leq p_k < 2^{2ut_k+1}$. Let $P_k = \prod_{i=1}^{k} p_i$ and $T_k = \sum_{i=1}^{k} t_i$. After receiving the redundancies $c_1, \ldots, c_k$ corresponding to $p_1, \ldots, p_k$, Neil has as much information as if Oscar would have transmitted a redundancy C_k modulo P_k. Oscar can indeed compute $S_k = C'_k/C_k$ from $s_k = c'_k/c_k$ and S_{k-1} using the Chinese Remainder Theorem (CRT):

$$\begin{aligned} S_k &= \mathrm{CRT}(S_{k-1}, P_{k-1}, s_k, p_k) \\ &= S_{k-1}(p_k^{-1} \bmod P_{k-1})p_k + s_k(P_{k-1}^{-1} \bmod p_k)P_{k-1} \bmod P_k. \end{aligned}$$

The full protocol is given in Figure 2 page 4. Note that no information is lost and that the transmitted modular knowledge about the difference adds up until it becomes sufficiently large to reconcile $\mathcal{H}$ and $\mathcal{H}'$. Therefore, the worst-case number of necessary rounds κ is the smallest integer k such that $T_k \geq T$.

In what follows, we will focus on two interesting choices of t_k:

- **Fixed t:** $\forall k,\ t_k = t$ for some fixed t, in which case $\kappa = \lceil \frac{T}{t} \rceil$;
- **Exponential t_k:** $\forall k,\ t_k = 2^k t$ for some fixed t, in which case $\kappa = \lceil \log_2 \frac{T}{t} \rceil$.

[3] *i.e.*, $\prod h'_i \mod a \neq 0$.

Oscar		**Neil**
Initial phase: Neil sends a global hash to detect $\perp_2$		
		$H \leftarrow \texttt{Hash}(\mathcal{H}')$
	$\xleftarrow{H}$	
Main phase: Neil amasses modular information on the difference between $\mathcal{H}$ and $\mathcal{H}'$		
	Round 1	
$c_1 \leftarrow \prod_{i=1}^{n} h_i \bmod p_1$	$\xrightarrow{c_1}$	$c_1' \leftarrow \prod_{i=1}^{n'} h_i' \bmod p_1$ $s_1 \leftarrow c_1'/c_1 \bmod p_1$ $S_1 \leftarrow s_1$ reconstruct a, b from $S_1 \ (\mathrm{mod} P_1)$ $\mathcal{D}' \leftarrow \{h_i' \in \mathcal{H}' \mid a \bmod h_i' = 0\}$ if $\prod_{h \in \mathcal{D}'} h \bmod P_1 = a$ then go to final phase else continue (event $\perp_1$)
	Round 2	
$c_2 \leftarrow \prod_{i=1}^{n} h_i \bmod p_2$	$\xrightarrow{c_2}$	$c_2' \leftarrow \prod_{i=1}^{n'} h_i' \bmod p_2$ $s_2 \leftarrow c_2'/c_2 \bmod p_2$ $S_2 \leftarrow \mathrm{CRT}(S_1, P_1, s_2, p_2)$ reconstruct a, b from $S_2 \ (\mathrm{mod} P_2)$ $\mathcal{D}' \leftarrow \{h_i' \in \mathcal{H}' \mid a \bmod h_i' = 0\}$ if $\prod_{h \in \mathcal{D}'} h \bmod P_2 = a$ then go to final phase else continue (event $\perp_1$)
	$\vdots$	
Final phase: Oscar obtains a candidate $\mathcal{H}'$ and hash-checks it		
	$\xleftarrow{\mathcal{D}', b}$	
$\mathcal{D} \leftarrow \{h_i \in \mathcal{H} \mid b \bmod h_i\}$ compute $\mathcal{H}'$ from $\{\mathcal{H}, \mathcal{D}, \mathcal{D}'\}$ if $\texttt{Hash}(\mathcal{H}') \neq H$ then go back to Main phase (event $\perp_2$) else the protocol is complete.		

Fig. 2. Complete D&F Protocol (for any T).

3 Transmission Complexity

This section proves that D&F achieves optimal asymptotic transmission complexity.

Assuming that no $\perp_2$ occurred (since $\perp_2$'s happen with negligible probability), D&F's transmission complexity is:

$$\sum_{k=1}^{\kappa} \log c_k + \log b + \log |\mathcal{D}'| \ \leq \ \sum_{k=1}^{\kappa} (ut_k + 1) + \frac{1}{2}(uT_\kappa) + uT \ \leq \ \frac{5}{2} uT_\kappa + u\kappa,$$

where κ is the required number of rounds.

For the two choices of t_k that we mentioned, transmission complexity is:

- **Fixed t:** $\kappa = \lceil T/t \rceil$, $T_\kappa = \kappa t < T + t$ and transmission is $\leq \frac{5}{2} u(T+t) + \lceil T/t \rceil = O(uT)$;
- **Exponential t:** $\kappa = \lceil \log(T/t') \rceil$, $T_\kappa < 2T$ and transmission is $\leq \frac{5}{2} \times 2uT + \lceil \log(T/t') \rceil = O(uT)$.

While asymptotic transmission complexities are identical for both choices, we note that the fixed t option is slightly better in terms of constant factors and halves transmission with respect to the exponential option. However, as we will see in Section 5.2, an exponential t results in a lower computational complexity.

Note that in both cases asymptotic transmission complexity is proportional to the size of the symmetric difference (*i.e.*, the number of differences times the size of an individual element). This is also the information-theoretic lower bound on the quantity of data needed to perform reconciliation. Hence, the protocol is asymptotically optimal from a transmission complexity standpoint.

Probabilistic Decoding: Reducing p. We now describe an improvement that reduces transmission by a constant factor at the expense of higher RNR failure rates. For simplicity, we will focus on one round D&F and denote by p the current P_k. We will generate a p about twice smaller than the p recommended in Section 2.2, namely $2^{ut-1} \leq p < 2^{ut}$.

Unlike Section 2.2, we do not have a fixed bound for a and b anymore; we only have a bound for the *product* ab, namely $ab \leq 2^{ut}$.

Therefore, we define $t+1$ couples of possible bounds: $(A_j, B_j)_{0 \leq j \leq t} = \left(2^{uj}, 2^{u(t-j)}\right)$.

Because $2^{ut-1} \leq p < 2^{ut}$ and $ab \leq 2^{ut}$, there must exist at least one index j such that $0 \leq a \leq A_j$ and $0 < b \leq B_j$. We can therefore apply Theorem 1 with $A = A_j$ and $B = B_j$: since $A_j B_j = 2^{ut} < p$, given (A_j, B_j, p, s), one can recover (a, b), and hence Oscar can compute $\mathcal{H}'$.

This variant will roughly halve transmission with respect to Section 2.2. The drawback is that, unlike Section 2.2, we have no guarantee that such an (a, b) is unique. Namely, we could in theory stumble over an $(a', b') \neq (a, b)$ satisfying the equation $a'b' \leq 2^{ut}$ for some index $j' \neq j$. We conjecture that, when u is large enough, such failures happen with a negligible probability (that we do not try to estimate here). This should lower the expected transmission complexity of this variant. In any case, thanks to hashing ($H = \texttt{Hash}(\mathcal{H}')$), if a failure occurs, it will be detected.

4 From Set Reconciliation to File Reconciliation

We now show how to perform *file reconciliation* using hashing and D&F. We then devise methods to reduce the size of hashes and thus improve transmission by constant factors. The presented methods are generic and can be applied to any set reconciliation protocol.

4.1 File Reconciliation Protocol

So far Oscar and Neil know how to synchronize sets of u-bit primes. They now want to reconcile files modeled as arbitrary length binary strings. Let $\mathcal{F} = \{F_1, \ldots, F_n\}$ be Oscar's set of files and let $\mathcal{F}' = \{F'_1, \ldots, F'_n\}$ be Neil's. Let $\eta = |\mathcal{F} \cup \mathcal{F}'| \leq n + n'$ be the total number of files.

A naïve way to reconcile $\mathcal{F}$ and $\mathcal{F}'$ is to simply hash the content of each file into a prime and proceed as before. Upon D&F's completion, Neil can send to Oscar the actual content of the files matching the hashes in $\mathcal{D}'$, *i.e.*, the files that Oscar does not have.

More formally, define for $1 \leq i \leq n$, $h_i = \texttt{HashPrime}(F_i)$ and for $1 \leq i \leq n'$, $h'_i = \texttt{HashPrime}(F'_i)$ where `HashPrime` is a collision-resistant hash function into primes so that the mapping from $\mathcal{F} \cup \mathcal{F}'$ to $\mathcal{H} \cup \mathcal{H}'$ is injective for all practical purposes. Section 5.1 shows how to construct such a hash function from usual hash functions.

4.2 The File Laundry: Reducing u

What happens if we brutally shorten u in the basic D&F protocol? As expected by the birthday paradox, we should start seeing collisions. In Appendix A, we analyze the statistics governing the appearance of collisions. The average number of colliding files is $\sim \eta(\eta - 1)2^{-u'}$ where $u' = u - \ln(u)$. For instance, the expected number of collisions for $\eta = 10^6$ and 42-bit digests, the average number of colliding files is < 4. We remark

that a collision can only yield a *false positive*, and never a *false negative*. In other words, while a collision may make Oscar and Neil miss a real difference, it will never create a nonexistent difference *ex nihilo*. Thus, it suffices to replace $\texttt{HashPrime}(F)$ by a diversified $\hbar_k(F) = \texttt{HashPrime}(k|F)$ to quickly filter-out file differences by repeating the protocol for $k = 1, 2, \ldots$ We call each complete D&F protocol repetition (which usually involves several basic protocol *rounds*) an *iteration*. At each iteration, the parties will detect files in $\mathcal{F} \Delta \mathcal{F}'$ whose hash $\hbar_{i,\ell}$ or $\hbar'_{i,\ell}$ does not collide, reconciliate these differences, remove these files from $\mathcal{F}$ and $\mathcal{F}'$ to avoid further collisions, and "launder" again the remaining files in the updated versions of $\mathcal{F}$ and $\mathcal{F}'$.

Let $\epsilon_{\eta,u,k}$ be the probability that at least one file will persist colliding during k rounds. Assuming that η is invariant between iterations, we find that $\epsilon_{\eta,u,k} \leq n((\eta-1)2^{-u'})^k$ *i.e.*, $\epsilon_{\eta,u,k}$ decreases exponentially in k (*e.g.*, $\epsilon_{10^6,42,2} \leq 10^{-3}\%$, see Appendix A.

We still need a condition to stop "laundering", *i.e.*, a condition ensuring that there are no more differences hidden by collisions. Before we describe this condition, let us first spell out the three kinds of collisions that can appear during iteration ℓ:

1. Collisions in $\mathcal{F} \cap \mathcal{F}'$ (*i.e.*, between common files). These are never a problem because they cannot hide any differences.
2. Collisions between between $\mathcal{F} \cap \mathcal{F}'$ and $\mathcal{F} \Delta \mathcal{F}'$ (*i.e.*, between a common file of Oscar and Neil, and a file not in common). These collisions can be easily detected by Oscar or Neil, at the end of iteration ℓ. However, if there is a collision of this kind involving an $h \in \mathcal{H} \Delta \mathcal{H}'$, we will not be able to find the file in $\mathcal{F} \Delta \mathcal{F}'$ matching h. For this reason, another D&F iteration will be necessary to reconcile this file.
3. Collisions in $\mathcal{F} \Delta \mathcal{F}'$ (*i.e.*, between files not in common). Such collisions hide real differences between $\mathcal{F}$ and $\mathcal{F}'$ and cannot be detected without a further iteration. This is why we need a condition to detect that no more collisions of this kind exist and stop laundering.

We propose the following method to decide termination. Before the first *iteration*, Neil sends a global hash $H' = \texttt{Hash}(\texttt{Hash}(F'_1), \ldots, \texttt{Hash}(F'_{n'}))$ to Oscar. This H' should not be confused with the H sent at the beginning of *each* iteration[4]. Now, if iteration ℓ is successful, Neil sends the list of $\texttt{Hash}(F'_i)$ for the new files $F'_i \in \mathcal{F}' \setminus \mathcal{F}$ whose hash $\hbar'_{i,\ell}$ does not collide with files in $\mathcal{F}' \cap \mathcal{F}$ (*i.e.*, type-2 collisions). Oscar can then use H' to check whether a type-3 collision remains, in which case a new iteration is performed. If no type-2 or type-3 collisions remain, then reconciliation is complete.

5 Computational Complexity

We now analyze D&F's computational complexity. We first describe the time complexity of a straightforward implementation (Section 5.1), and then present four independent optimizations (Section 5.2). A summary of all costs is given in Table 1. To simplify analysis, we assume that there are no collisions, and that $n = n'$.

5.1 Basic Complexity and Hashing Into Primes

Let $\mu(\ell)$ be the time required to multiply two ℓ-bit numbers, with the assumption that $\forall \ell, \ell', \mu(\ell + \ell') \geq \mu(\ell) + \mu(\ell')$. For naïve (*i.e.*, convolutional) algorithms, $\mu(\ell) = O(\ell^2)$, but using FFT multiplication [12], $\mu(\ell) = \tilde{O}(\ell)$ (where $\tilde{O}(f(\ell))$ is a shorthand for $O(f(\ell)\log^k f(\ell))$ for some k). FFT is experimentally faster than convolutional methods from $\ell \sim 10^6$ and on. The modular division of a 2ℓ-bit number by an ℓ-bit number and the reduction of a 2ℓ-bit number modulo an ℓ-bit number are also known to cost $\tilde{O}(\mu(\ell))$ [3]. Indeed, in packages such as GMP, division and modular reduction run in $\tilde{O}(\ell)$ for sufficiently large ℓ.

The naïve complexity of $\texttt{HashPrime}$ is $u^2\mu(u)$, as per [8,2].

- A recommended implementation of $\texttt{HashPrime}(F)$ consists in defining the digest as $h = 2 \cdot \texttt{Hash}(F|i) + 1$ and increasing i until h is prime. Because there are roughly $\frac{2^u}{u}$ u-bit primes we need to perform (on average) u primality tests before finding a suitable h. The cost of a Miller-Rabin primality test is $\tilde{O}(u\mu(u))$. Hence, the total cost of this implementation is $\tilde{O}(u^2\mu(u))$. A more precise analysis can be found in [2].

[4] H is a hash of the (potentially colliding) diversified hashes $\hbar(\ell|F)$.

– If u is large enough (*e.g.*, 160) one might sacrifice uniformity to avoid repeated file hashings using $\texttt{HashPrime}(F) = \texttt{NextPrime}(\texttt{Hash}(F))$.
– Yet another acceleration option consists in computing $h = \alpha \lfloor \texttt{Hash}(F)/\alpha \rfloor + 1$, where $\alpha = 2 \times 3 \times 5 \times \cdots \times \texttt{Prime}[d]$ is the product of the first primes until some rank d, and then subtract α from h until h becomes prime. Denote by ϕ Euler's totient function and assume that this algorithm randomly samples u-bit numbers congruent to 1 mod α until it finds a prime. There are about $\frac{2^u}{u\phi(\alpha)}$ u-bit primes congruent to 1 mod α, and there are $\frac{2^u}{\alpha}$ u-bit numbers congruent to 1 mod α. Thus, the algorithm is expected to do about $\frac{2^u}{\alpha}/\frac{2^u}{u\phi(\alpha)} = \frac{\phi(\alpha)}{\alpha}u$ primality tests before finding a prime, which improves over the u tests required by the naïve algorithm. The main drawback of this algorithm is that, even if `Hash` is uniformly random, `HashPrime` isn't. This slightly increases `HashPrime`'s collision-rate and u has to be increased subsequently.

5.2 Optimizations

Adapting p_k. Taking the p_k's to be ut_k-bit primes is inefficient, because large prime generation is slow. In this section, we study alternative p_k choices yielding constant factor improvements.

Let $\texttt{Prime}[i]$ denote the i-th prime, with $\texttt{Prime}[1] = 2$. Besides conditions on size, the *only* property required from a p_k is to be co-prime with the $\{h_i, h'_i\}$. We can hence consider the following variants, all which will imply a few conditions on $\{h_i, h'_i\}$ to ensure this co-primality:

– *Variant 1.* $p_k = \prod_{j=r_k}^{r_{k+1}-1} \texttt{Prime}[j]$ where the bounds r_k are chosen to ensure that each p_k has the proper size. Generating such smooth numbers is much faster than generating large primes.
– *Variant 2.* $p_k = \texttt{Prime}[k]^{r_k}$ where the exponents r_k are chosen to ensure that each p_k has the proper size. This is faster than Variant 1 and requires that $\min\{h_i, h'_i\} > \max(\texttt{Prime}[k])$.
– *Variant 3.* $P_k = 2^{ut_k}$. In this case $C_k = \prod_{i=1}^{n} h_i \bmod P_k$, $c_1 = C_1$ and $c_k = (C_k - C_{k-1})/P_{k-1}$. *i.e.*, c_k is the slice of bits $ut_{k-1}, \ldots, ut_k - 1$ of C_k denoted $c_k = C_k[ut_{k-1}, \ldots, ut_k]$. Variant 3 is modular-reduction-free and CRT-free: C_k is just the binary concatenation of c_k and C_{k-1}. Computations are thus much faster. Algorithm 1 (justified hereafter) computes c_k efficiently. Note that we only need to store $D_{k,i}$ and $D_{k+1,i}$ during round k (for all i). So space overhead is $O(nu)$.

  Let $X_i = \prod_{j=1}^{i} h_j$ (with $X_0 = 1$), $X_{i,k} = X_i[ut_{k-1} \cdots ut_k]$ and let $D_{i,k}$ be the u most significant bits of the product of $Y_{i,k} = X_i[0 \cdots ut_k]$ and h_i, *i.e.*, $D_{i,k} = (Y_{i,k} \times h_i)[ut_k \cdots u(t_k + 1)]$ (with $D_{i,0} = 0$ and $D_{0,k} = 0$). Since $X_{i+1} = X_i \times h_{i+1}$, we have, for $k \geq 0$, $i \geq 0$:

$$\begin{aligned} D_{i+1,k+1} \times 2^{ut_{k+1}} + Y_{i+1,k+1} &= Y_{i,k+1} \times h_{i+1} \\ &= (X_{i,k+1} \times 2^{ut_k} + Y_{i,k}) \times h_{i+1} \\ &= X_{i,k+1} \times 2^{ut_k} \times h_{i+1} + Y_{i,k} \times h_{i+1} \\ &= X_{i,k+1} \times h_{i+1} \times 2^{ut_k} + (D_{i,k} \times 2^{ut_k} + \cdots). \end{aligned}$$

  Therefore, if we only consider bits $[ut_k \cdots u(t_{k+1} + 1)]$, for $k \geq 1$, $i \geq 0$:

$$D_{i+1,k+1} \times 2^{u(t_{k+1}-t_k)} + X_{i+1,k+1} = X_{i,k+1} \times h_{i+1} + D_{i,k}.$$

  Since $c_k = X_{n,k}$, Algorithm 1 is correct.

Algorithmic Optimizations using Product Trees. The non-overwhelming (but nonetheless important) complexities of the computations of (c, c') and of the factorizations can be even reduced to $\tilde{O}(\frac{n}{t_k}\mu(ut_k))$ and $\tilde{O}(\frac{n}{T_k}\mu(uT_k))$ as explained in Appendix B.

Algorithm 1 Computation of c_k for $p_k = 2^{ut_k}$

Require: k, the set h_i, $(D_{k,i})$ as explained in Appendix B
Ensure: $c_{k+1} = \prod_{i=1}^{n} h_i \bmod p_{k+1}$, (D_{k+1}) as explained in Appendix B
1: **if** $k = 0$ **then** $X \leftarrow 1$ **else** $X \leftarrow 0$
2: **for** $i = 1, \ldots, n$ **do**
3: $\quad Z \leftarrow X \times h_i$
4: $\quad D_{i,k+1} \leftarrow Z[u(t_{k+1} - t_k) \cdots u(t_{k+1} - t_k + 1)]$
5: $\quad X \leftarrow Z[0 \cdots u(t_{k+1} - t_k)]$
6: $c_{k+1} \leftarrow X$

Table 1. Protocol Complexity Synopsis

Entity	Computation	Complexity in $\tilde{O}$ of		Optimization
		Basic algo.[a]	Opt. algo.[b]	
Both	computation of h_i and h'_i	$nu^2 \cdot \mu(u)$	$\frac{\phi(\alpha)}{\alpha} nu^2 \cdot \mu(u)$	fast hashing
	Round k			
Both	compute redundancies c_k and c'_k	$n \cdot \mu(ut_k)$	$\frac{n}{t_k} \cdot \mu(ut_k)$	prod. trees
Neil	compute $s_k = c'_k/c_k$[c] or $S_k = C'_k/C_k$[d]	$\mu(uT_k)$	$\mu(uT_k)$	
Neil	compute S_k from S_{k-1} and s_k (CRT)[c]	$\mu(uT_k)$	none	$p_k = 2^{ut_k}$
Neil	find a_k, b_k such that $S_k = a_k/b_k \bmod P_k$[e]	$\mu(uT_k)$	$\mu(uT_k)$	
Neil	factor a_k	$n \cdot \mu(uT_k)$	$\frac{n}{T_k} \cdot \mu(uT_k)$	prod. trees
	Last round			
Oscar	factor b_k	$n \cdot \mu(ut_k)$	$\frac{n}{t_k} \cdot \mu(ut_k)$	prod. trees
	global complexity			
	...with naïve mult.	$nu^2T^3/t + nu^4$	$nu^2T + \frac{\phi(\alpha)}{\alpha} nu^4$	doubling[f]
	...with FFT	$nuT + nu^3$	$nu + \frac{\phi(\alpha)}{\alpha} nu^3$	doubling[f]

[a] using the basic algorithms of Section 5.1, and taking $t_1 = t_2 = \cdots = t$
[b] using all the optimizations of Sections 5.1, 5.2 ($p_k = 2^{ut_k}$ also yields substantial constant factor accelerations not shown in this table), the product trees and the doubling as described in the full version of this paper
[c] only for prime p_i (or variant 1 or 2 in Section 5.2)
[d] only for $p_i = 2^{ut_i}$
[e] using advanced algorithms in [11,15] — naïve extended GCD gives $(uT_i)^2$
[f] in addition to the previous optimizations

Doubling. As seen at the end of Section 2.3, using the exponential t variant (*i.e.*, doubling t_k at each iteration) doubles transmission (at most) with respect to the fixed t option, but drastically reduces the amount of computation to perform.

6 Related Work on Set Reconciliation

This section compares D&F with the set reconciliation algorithm of Minsky *et alii* [10] (hereafter MTZ). We do not analyze here reconciliation algorithms achieving better computational performances at the cost of supra-linear transmission complexity (*e.g.*, [6] or [4]).

Unlike MTZ which is based on polynomials, D&F is based on integers. D&F and MTZ both achieve an optimal (*i.e.*, linear) transmission complexity, but D&F only deals with fixed-size primes, whereas MTZ deals with any fixed-size bit strings.

MTZ is mostly designed for "incremental" settings where $\mathcal{H}$ and $\mathcal{H}'$ are often updated[5] and re-synchronized. This differs from our setting and there seems to be no straightforward manner to extend D&F in that fashion while maintaining a low time complexity. For that reason, our analysis of MTZ's time complexity will take into account the cost of computing redundancies, as we did for D&F. The main differences between MTZ and D&F are the following:

- MTZ synchronizes monic polynomials $X - h_i$ and $X - h'_i$ over a field $\mathbb{F}_q$ (where q is a $(u+1)$-bit prime), instead of u-bit primes $\{h_i, h'_i\}$;
- In MTZ, p_k are square-free, mutually co-prime polynomials which are also co-prime with all $X - h_i$ and $X - h'_i$. In D&F this role is played by mutually co-prime integers that are also co-prime with respect to the $\{h_i, h'_i\}$ (for all the variants in Section 5.2 except the last).

Indeed, in the basic one-round version:

- If we write $p_k = (X - \rho_1) \cdots (X - \rho_t)$, then $c = \prod_{i=1}^{n}(X - h_i) \bmod p_k$ and $\chi_{\mathcal{H}} = \prod_{i=1}^{n}(X - h_i)$, $\chi_{\mathcal{H}}(\rho_j) = c(\rho_j)$. Thus, thanks to Lagrange interpolation, sending evaluations of $\chi_{\mathcal{H}}$ in t points $\rho_1, \ldots, \rho_j$, as Oscar does in [10], is equivalent to sending c.
- The rational function interpolation of [10] can also be seen as an RNR version of Theorem 1 for polynomials: we try to recover two polynomials a, b (with a correct bound on degrees) such that $ab^{-1} \bmod p = c'c^{-1} \bmod p$. Note that this implies that the Gaussian elimination of cost $O(t^3\mu(u))$ (used for this step by MTZ) can be replaced by an extended GCD computation that costs only $O(t^2\mu(u))$ (and $\tilde{O}(\mu(ut))$ using the advanced algorithms of [11,15]);

We will compare the computational complexities of MTZ and D&F without taking into account the cost of hashing the files that has to be incurred by both algorithms. MTZ's time complexity is thus $O(nu^2T + u^2T^3)$ when doubling is used, which is not as good as our $\tilde{O}(nu)$ with FFT, and also not as good as our non-FFT complexity $\tilde{O}(nu^2T)$ when $n \ll T^2$. However, our better complexity bounds stem from optimizations that are all equally applicable to MTZ (except, of course, the optimizations concerning the choice of p_k).

An improved way to perform set reconciliation is presented in [9]. This algorithm uses MTZ as a black box and requires at least about $24e \cong 65.23$ times more bandwidth (with a bipartition) but substantially improves MTZ's computational complexity. However, this construction is generic with respect to the underlying reconciliation algorithm and can hence be applied to D&F to yield identical complexity gains.

7 From File Reconciliation to File Synchronization

In Section 4, we reconciled file sets by looking only at their contents. However, in practice, users synchronize file *sets*, and not just *hierarchies*. In other words, we are not just interested in file *contents* but also in their *metadata*. The most important metadata is the file's path (*i.e.*, its name and location in the filesystem), though other kinds of metadata exist (*e.g.*, modification time, owner, permissions). In many cases, file metadata change while the file contents do not: *e.g.*, files can be *moved* to a different directory. When performing reconciliation, we must be aware of this fact, and reflect file moves without re-transferring the moved files' contents. (This is an important improvement over popular synchronization tools such as `rsync`).

We will call this task *file synchronization*. This section achieves *file synchronization* using D&F as a black-box. The described algorithms are hence generic and can leverage any reconciliation algorithm.

7.1 General Principle

To perform file synchronization, Oscar and Neil will hash the contents of each of their files using a collision-resistant hash function `Hash`: we will call this the file's *content hash* and denote it by C_i or C'_i for the i-th file in $\mathcal{F}$ or $\mathcal{F}'$. Likewise, we denote by M_i or M'_i the files' metadata. We let F_i or F'_i denote the pair (C_i, M_i) or (C'_i, M'_i). Oscar and Neil will reconciliate those sets as in Section 4.

Once the reconciliation has completed, Oscar is aware of the metadata and the content hash of all of Neil's files that do not exist in his disk with the same content and metadata (we will call these the *missing files*).

[5] *e.g.*, by adding or removing a few values to $\mathcal{H}$ or $\mathcal{H}'$.

Oscar now looks at the list of the missing files' content hashes. For some of these hashes, Oscar may already have a file with the same content hash, but only with a wrong metadata. For others, Oscar may not have any file with the same content hash. In the first case, Oscar can recreate Neil's file by altering the metadata, without retransferring the file's contents. This is presented in Section 7.2. In the second case, Oscar needs to retrieve the full file contents from Neil. This is presented in Section 7.3.

7.2 Moving Existing Files

Adjusting the metadata of existing files is trivial, except for file location which is the focus of this section: Oscar needs to perform a sequence of file moves on his copy to reproduce the structure of Neil's copy. 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, but b already exists), or because a folder cannot be created (we want to move a to b/c, but b already exists as a file and not as a folder). Note that for a move operation $a \to b$, there is at most one file blocking the location b: we will call it the *blocker*.

If the blocker is absent on Neil, then we can just delete the blocker. However, if a blocker exists and is a file which appears in Neil with different metadata, then we might need to move this blocker somewhere else before we apply the move we are interested in. Moving the blocker might be impossible because of another blocker that we need to keep, and so on, possibly ending in a cycle (*e.g.*, move a to b and b to a) in which case we need to use an intermediate temporary location.

How should we perform the moves? A simple way would be to move each file to a unique temporary location and then move them to their final location: however, this performs many unnecessary moves and could lead to problems if the process is interrupted. We can do something more clever by performing a decomposition into Strongly Connected Components (SCC) of the *move graph* (with one vertex per file and one edge per move operation going from to the file to its blocker or to its destination if no blocker exists).

Once the SCC decomposition is known, moves can be applied by performing them in 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 2.

7.3 Transferring Missing Files

Once all moves have been applied, Oscar's hierarchy contains all of its files which also exist on Neil. These have been put at the correct location and have the right metadata. The only thing that remains is to transfer the contents of Neil's files that do not exist in Oscar's hierarchy and create those files at the right position. To do so, we can just use `rsync` to synchronize explicitly the correct files on Neil to the matching locations in Oscar's hierarchy, using the fact that Oscar is now aware of all of Neil's files and their locations. In so doing, we have to ensure that multiple files on Neil that have the same content are only transferred once and then copied to all their locations without being retransferred.

It is interesting to notice that if a file's contents has been changed slightly on Neil but its location hasn't changed, then in most cases the `rsync` invocation will reuse the existing copy of the file on Oscar when transferring this file from Neil to Oscar. Because `rsync` uses rolling checksums to retransfer only relevant file parts, this may actually reduce the transmission complexity. If a file's content is slightly changed and the file is moved, however, then this gain will not occur.

8 Implementation

We implemented D&F, extended it to perform file synchronization, and benchmarked it against `rsync`. The implementation is called `btrsync`, its source code is available from [1]. `btrsync` was written in Python (using GMP to perform the number theoretic operations), and uses a bash script (invoking SSH) to create a secure communication channel between Oscar and Neil.

Algorithm 2 Perform moves

Require: $\mathfrak{M}$ is a dictionary where $\mathfrak{M}[f]$ denotes the intended destinations of f

$C \leftarrow []$ ▷ Stores the *color* of a file (initially "not_done")
$T \leftarrow []$ ▷ Stores the temporary location assigned for a file
function UNBLOCK_COPY(f, d)
    if d is blocked by some b **then**
        if b is not in $\mathfrak{M}$'s keys **then** delete(b) ▷ We don't need b
        else RESOLVE(b) ▷ Take care of b and make it go away
    if $T[f]$ was set **then** $f \leftarrow T[f]$
    copy(f, d)
function RESOLVE(f)
    if $C[f] =$ done **then**
        return ▷ Already managed by another in-edge
    if $C[f] =$ doing **then**
        if $T[f]$ was not set **then**
            $T[f] \leftarrow$ mktemp() ▷ Use a new temporary location
            move(f, $T[f]$)
        return ▷ We found a loop, moved f out of the way
    $C[f] \leftarrow$ doing
    for $d \in \mathfrak{M}[f]$ with $d \neq f$ **do**
        UNBLOCK_COPY(f, d) ▷ Perform all the moves
    if $f \notin \mathfrak{M}[f]$ and $T[f]$ was not set **then** delete(f)
    if $T[f]$ was set **then** delete($T[f]$)
    $C[f] \leftarrow$ done
for f in $\mathfrak{M}$'s keys **do**
    RESOLVE(f)

Table 2. Test Directories

Directory	Description
`syn`	a directory containing 1000 very small files
`syn_shuf`	`syn` changed by 10 deletions, renames and modifications
`source`	a snapshot of `btrsync`'s own source tree
`source_moved`	`source` with one big folder (a few megabits) renamed
`ff-13.0`	the source archive of Mozilla Firefox 13.0
`ff-13.0.1`	the source archive of Mozilla Firefox 13.0.1
`empty`	an empty folder

8.1 Implementation Choices

Our implementation does not take into account all the possible optimizations described in Section 5: it implements doubling (Section 5.2) and uses powers of small primes for the p_k (variant 2 of Section 5.2), but does not implement product trees (Section 5.2) nor does it use the prime hashing scheme (Section 5.1). Besides, we did not implement the proposed improvement in transmission complexity for file reconciliation (Section 4.2).

As for file synchronization (Section 7), the only metadata managed by `btrsync` is the file's path (name and location). Other metadata types (modification date, owner, permissions) are not implemented, although it would be very easy to do so. An optimization implemented by `btrsync` over the move resolution algorithm described in Section 7.2 is to avoid doing a copy of a file F and then removing F: the implementation replaces such operations by moves, which are faster than copies on most file systems because the OS does not need to copy the actual file contents.

8.2 Experimental Comparison to `rsync`

We compared `rsync`[6] and our implementation `btrsync`. The directories used for the benchmark are described in Table 2. Experiments were performed without any network transfer, by synchronizing two folders on the same host. Hence, time measurements mostly represent the synchronization's CPU cost.

Results are given in Table 3. In general, `btrsync` spent more time than `rsync` on computation (especially when the number of files is large, which is typically seen in the experiments involving `syn`). Transmission results, however, are favorable to `btrsync`.

In the trivial experiments where either Oscar or Neil have no data at all, `rsync` outperforms `btrsync`. This is especially visible when Neil has no data: `rsync`, unlike `btrsync`, immediately notices that there is nothing to transfer.

In non-trivial tasks, however, `btrsync` outperforms `rsync`. This is the case of the `syn` datasets, where `btrsync` does not have to transfer information about all unmodified files, and even more so in the case where there are no modifications at all. For Firefox source code datasets, `btrsync` saves a very small amount of bandwidth, presumably because of unmodified files. For the `btrsync` source code dataset, we notice that `btrsync`, unlike `rsync`, was able to detect the move and avoid retransferring the moved folder.

9 Conclusion and Further Improvements

This paper introduced the new number-theoretic set reconciliation protocol called *Divide and Factor* (D&F). We analyzed D&F's transmission and time complexities and describing several optimizations and parameter

[6] `rsync` version 3.0.9, used both as a competitor to benchmark against and as an underlying call in our own code. `rsync` was passed the following options: `--delete` to delete Oscar's files that were deleted on Neil like `btrsync` does, `-I` to disable heuristics based on file modification times that `btrsync` does not use, `--chmod="a=rx,u+w"` to make it unnecessary to transfer file permission that `btrsync` does not transfer (though verbose logging suggest that `rsync` wastes a few bytes per file because it transmits them anyway), and `-v` to count the number of sent and received bytes.

Table 3. Experimental results. Synchronization is performed *from* Neil *to* Oscar. `RX` and `TX` denote the quantity of received and sent bytes, `rs` and `bt` denote `rsync` and `btrsync`, and $\delta_\square = \texttt{TX}_\square + \texttt{RX}_\square$. $\delta_{\texttt{rs}} - \delta_{\texttt{bt}}$ and $\delta_{\texttt{bt}}/\delta_{\texttt{rs}}$ express the absolute and the relative differences in transmission between `rsync` and `btrsync`. The last two columns show timing results on an Intel Core i3-2310M CPU clocked at 2.10 Ghz.

Entities and Datasets		**Transmission (Bytes)**						**Time (s)**	
Neil's $\mathcal{F}'$	**Oscar's $\mathcal{F}$**	$\texttt{TX}_{\texttt{rs}}$	$\texttt{RX}_{\texttt{rs}}$	$\texttt{TX}_{\texttt{bt}}$	$\texttt{RX}_{\texttt{bt}}$	$\delta_{\texttt{bt}} - \delta_{\texttt{rs}}$	$\frac{\delta_{\texttt{bt}}}{\delta_{\texttt{rs}}}$	$\texttt{t}_{\texttt{rs}}$	$\texttt{t}_{\texttt{bt}}$
`source`	`empty`	778k	2k	780k	10k	10k	1.0	0.1	0.7
`empty`	`source`	24	12	12k	6k	18k	496.6	0.0	0.4
`empty`	`empty`	24	12	19	30	13	1.4	0.0	0.3
`syn`	`syn_shuf`	55k	19k	7k	3k	-63k	0.1	0.5	0.8
`syn_shuf`	`syn`	54k	19k	7k	3k	-63k	0.1	0.2	0.8
`syn`	`syn`	55k	19k	327	30	-73k	0.0	0.5	0.7
`ff-13.0.1`	`ff-13.0`	41M	1k	40M	3k	-1M	1.0	1.6	8.1
`source_moved`	`source`	778k	1k	3k	2k	-775k	0.0	0.1	0.4

choices. We have shown how D&F can be applied to file reconciliation using hashing, and to solve the practical file synchronization problem. D&F was benchmarked against `rsync`. The comparison reveals that D&F transmits less data than `rsync` but performs more computation.

Many interesting problems are left open. These problems are both theoretical and practical. A first theoretical challenge consists in eliminating the costly hashing into primes. *e.g.*, if the p_k are powers of two then hashing into odd integers might suffice. This would make reconciliation harder because multiple factorizations of a and b as products of h_i and h'_i could exist while only one of them would be the correct one. A careful probabilistic analysis would be required to determine the probability of multiple factorizations and bound the cost of recovering the correct factorization. This phenomenon is tightly linked to the cryptographic notion of *collision-division* [5]. As for other aspects of our construction, many bounds on transmission and computational complexities could be refined and improved.

Other theoretical questions are left open by our study of move resolution: The algorithm that we propose is *suboptimal* because there should never be any need to use two different temporary file locations: one location is always sufficient to break cycles, and a more careful exploration of the move graph could proceed in that fashion. It is also interesting to find out if there is a way to perform a minimal number of temporary moves (or if this problem is NP-complete), or if we can reduce the total number of moves by moving folders in addition to files.

From a practical standpoint, our `btrsync` implementation could be improved in several ways. First, the numerous possible improvements described in the paper could be implemented and benchmarked. Then, heuristics could be added to work around the situations in which `btrsync` is outperformed by `rsync`, such as the ones identified during our experimental comparison of the two programs. For instance, whenever the product of Neil's hashes becomes smaller than P_k, then Neil should send its hashes immediately to Oscar and terminate the protocol: this would avoid transmitting a lot of data in situations where Neil's copy is empty or very small. Last but not least, the development of our `btrsync` prototype could be continued to make it suitable for real-world users, including proper management of all metadata, using the file modification time as a heuristic to detect changes, and caching of file content hashes to avoid recomputing them.

A possible additional feature that could be added to `btrsync` is to detect files that have been both moved and altered slightly. A related improvement would be to use a variant of `rsync`'s algorithm to transfer Neil's new files to Oscar by considering simultaneously several related files on Oscar's copy and computing rolling checksums.

Finally, we could study how additional information could be used to speed up set reconciliation. An interesting possibility is to give to Neil and Oscar, in addition to their files, a value for each file indicating the probability that the other party does not have this file. To what extent could this prior knowledge be exploited to perform reconciliation more efficiently?

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

References

1. `https://github.com/RobinMorisset/Btrsync`
2. Abdalla, M., Ben Hamouda, F., Pointcheval, D., *Tighter Reductions for Forward-Secure Signature Schemes*, Accepted to PKC 2013 to appear in LNCS, Springer, 013. Full version available from the authors' webpage.
3. Burnikel, C., Ziegler, J., Stadtwald, I., *Fast Recursive Division*, Tech. Rep., MPI-I-98-1-022, MPI Informatik Saarbrucken, 1998.
4. Byers, J., Considine, J., Mitzenmacher, M., Rost, S., *Informed Content Delivery Across Adaptive Overlay Networks*, ACM SIGCOMM Computer Communication Review, vol. 32(4), pp. 47–60, 2002.
5. Coron, J.-S., Naccache, D., *Security Analysis of The Gennaro-Halevi-Rabin Signature Scheme*, EUROCRYPT'00, LNCS vol. 1807, Springer, pp. 91–101, 2000.
6. Eppstein, D., Goodrich, M., Uyeda, F., Varghese, G., *What's the Difference?: Efficient Set Reconciliation Without Prior Context*, ACM SIGCOMM Computer Communication Review, vol. 41, pp. 218–229. 2011.
7. Fouque, P.A., Stern, J., Wackers, J.G., *Cryptocomputing With Rationals*, Financial Cryptography'02. LNCS vol. 2357, Springer, pp. 136–146, 2002.
8. Hohenberger, S., Waters, B., *Short and Stateless Signatures From the RSA Assumption*, CRYPTO'09. LNCS vol. 5677, Springer, pp. 654–670, 2009.
9. Minsky, Y., Trachtenberg, A., *Practical Set Reconciliation*, Tech. Rep., Department of Electrical and Computer Engineering, Boston University, Technical Report BU-ECE-2002-01, 2002, a full version can be downloaded from `http://ipsit.bu.edu/documents/BUTR2002-01.ps`
10. Minsky, Y., Trachtenberg, A., Zippel, R., *Set Reconciliation With Nearly Optimal Communication Complexity*, IEEE Transactions on Information Theory, vol. 49(9), pp. 2213–2218, 2003.
11. Pan, V., Wang, X., *On Rational Number Reconstruction and Approximation* SIAM Journal on Computing vol. 33(2), pp. 502–503, 2004.
12. Schönhage, A., Strassen, V., *Schnelle Multiplikation großer Zahlen*. Computing vol. 7(3), pp. 281–292, 1971.
13. Tridgell, A., *Efficient Algorithms for Sorting and Synchronization*, Ph.D. thesis, The Australian National University, 1999.
14. Vallée, B., *Gauss' Algorithm Revisited*. Journal of Algorithms vol. 12(4), pp. 556–572, 1991.
15. Wang, X., Pan, V., *Acceleration of Euclidean Algorithm and Rational Number Reconstruction*, SIAM Journal on Computing vol. 32(2), pp. 548–556, 2003.

A The File Laundry: Reducing u

Naïve Analysis. Let us now bound the number of iterations to perform, in a naïve manner. We slightly change notations and write:

$$\mathcal{F} \cup \mathcal{F}' = \{F_1, \ldots, F_\eta\}.$$

We notice that, as soon as a file F_i or F'_i in $\mathcal{F} \Delta \mathcal{F}'$ does not collide with another file in one iteration of D&F, then this difference between $\mathcal{F}$ and $\mathcal{F}'$ will be reconciled. In this naïve analysis, we do not use the fact that such files are removed from one iteration to the next, and we just compute the probability that a file keeps colliding for λ iterations. We will thus obtain an upper bound on the probability that at least one difference between $\mathcal{F}$ and $\mathcal{F}'$ has not been detected after λ iterations (*i.e.*, at least one false positive survives λ iterations).

Because there are are about $\frac{2^u}{u}$ u-bit prime numbers, we write $u' = u - \ln(u)$ and see `HashPrime` as a random function from $\{0,1\}^*$ to $\{0, \ldots, 2^{u'} - 1\}$. Let X_i be the random variable:

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

Clearly, we have $\Pr[X_i = 1] \leq \frac{\eta-1}{2^{u'}}$. Using this bound and the linearity of expectation, the average number of colliding files is hence:

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

For instance, for $\eta = 10^6$ files and 42-bit digests, the expected number of colliding files is less than 4.

Assume that the diversified $\hbar_\ell(F)$'s are random and independent. We will show that the probability that a stubborn file continues to collide decreases exponentially with the number of iterations λ. Assume that η remains invariant between iterations and define the following random variables:

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

$$Y_i = \bigwedge_{\ell=1}^{\lambda} X_i^\ell = \begin{cases} 1 & \text{if file } F_i \text{ collides during all the } \lambda \text{ first iterations.} \\ 0 & \text{otherwise.} \end{cases}$$

By independence, we have:

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

Therefore the average number of colliding files is:

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

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

$$\epsilon_\lambda \leq \eta\left(\frac{\eta-1}{2^{u'}}\right)^\lambda$$

For the previously considered instance of $\eta = 10^6$ and $u = 42$, we get $\epsilon_2 \leq 10^{-3}\%$, and with probability more than $1 - \epsilon_2$, two iterations of D&F will suffice.

How to Select u? For the sake of simplicity, we consider $t_1 = t_2 = \cdots = t$. For a fixed λ, ϵ'_λ decreases as u' grows. For a fixed u', ϵ'_λ also decreases as λ grows. Transmission, however, grows with both u' (bigger digests) and k (more iterations). We write for the sake of clarity: $\epsilon'_\lambda = \epsilon'_{\lambda,u',\eta}$.

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

$$T_{u,\eta} \propto u\sum_{\lambda=1}^{\infty} \lambda \cdot \epsilon'_{\lambda,u',\eta} = \frac{u'\eta(\eta-1)}{2^{u'}} \sum_{\lambda=1}^{\infty} \left(\frac{\eta-1}{2^{u'}}\right)^{\lambda-1} = \frac{u\eta(\eta-1)}{2^{u'}-\eta+1} = \frac{u\eta\,(\eta-1)\,8^u}{\left(2^u - 4^u + (\eta-2)^2\right)^2}$$

Dropping the proportionality factor $\eta(\eta-1)$ and approximating $\eta - 1 \approx \eta$, we can optimize the function:

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

$\phi_{10^6}(u')$ admits an optimum for $u' \approx 15$.

Possible Refinements. The previous analyses are incomplete because of the following approximations:

- We used a fixed u in all rounds. Nothing forbids using a different u_ℓ at each iteration, or even fine-tuning the u_ℓ's adaptively as a function of the laundry's effect on the progressively reconciliated multisets..
- Our analysis treats t as a constant, but large t values increase p and hence the number of potential files detected as different per iteration - an effect disregarded *supra*.

A different approach is to optimize t and u experimentally, *e.g.*, using the open source D&F program `btrsync` developed by the authors (*cf.* Section 8).

B Algorithmic Optimizations using Product Trees

For the sake of simplicity, we suppose there is only one round, write $p = p_1$, $t = t_1 = T_1$, and assume that t is a power of two dividing n. We write $t = 2^\tau$. For simplicity, we also assume that p is prime, though the algorithm can be easily adapted to the other variants described in Section 5.2.

The idea is the following: group h_i's by subsets of t elements and compute the product of each such subset in $\mathbb{N}$ (*i.e.*, without modulo). For $j \in \{1, \ldots, n/t\}$, we define:

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

Each H_j can be computed in $\tilde{O}(\mu(ut))$ using the standard product tree method described in Algorithm 3. (for $j = 1$). Thus, all these $\frac{n}{t}$ products can be computed in $\tilde{O}(\frac{n}{t}\mu(ut))$. We can then compute c by multiplying the H_j modulo p, which costs $\tilde{O}(\frac{n}{t}\mu(ut))$.

Algorithm 3 Product tree algorithm (assuming $j = 1$ for simplicity).

Require: the set h_i
Ensure: $\pi = \pi_1 = \prod_{i=1}^{t} h_i$, and π_i for $i \in \{1, \ldots, 2t-1\}$
1: $\pi \leftarrow$ array of size t
2: **function** PRODTREE(i, start, end)
3: **if** start = end **then**
4: **return** 1
5: **else if** start + 1 = end **then**
6: **return** $h_{\text{start}+1}$
7: **else**
8: mid $\leftarrow \lfloor \frac{\text{start+end}}{2} \rfloor$
9: $\pi_{2i} \leftarrow$PRODTREE($2i$, start, mid)
10: $\pi_{2i+1} \leftarrow$PRODTREE($2i+1$, mid, end)
11: **return** $\pi_{2i} \times \pi_{2i+1}$
12: $\pi_1 \leftarrow$PRODTREE($1, 0, t$)

The same technique applies to factorization. We explain the process with a, though the same process applies *ne variatur* to b. After computing the tree product, we can compute the residues of a modulo H_1. Then we can compute the residues of a mod H_1 modulo the two children π_2 and π_3 of $H_1 = \pi_1$ in the product tree (depicted in Figure 3), and so on. Intuitively, we descend the product tree doing modulo reduction. At the end (*i.e.*, as we reach the leaves), we obtain the residues of a modulo each of the h_i ($i \in \{1, \ldots, t\}$). This is described in Algorithm 3. We can use the same method for the tree product associated to any H_j, and the residues of a modulo each of the h_i ($i \in \{(j-1)t+1, \ldots, jt\}$) for any j, *i.e.*, a modulo each of the h_i for any i. Complexity is $\tilde{O}(\mu(ut))$ for each j, which amounts to a total complexity of $\tilde{O}(\frac{n}{t}\mu(ut))$.

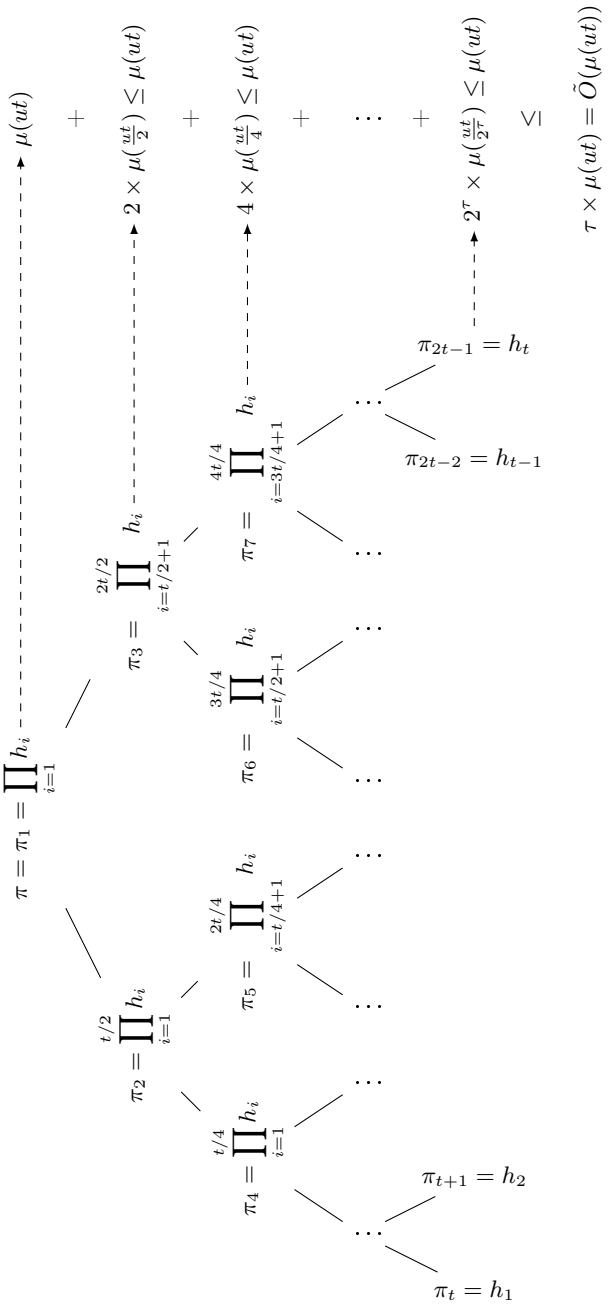

Fig. 3. Product tree (assuming $j = 1$ for simplicity).

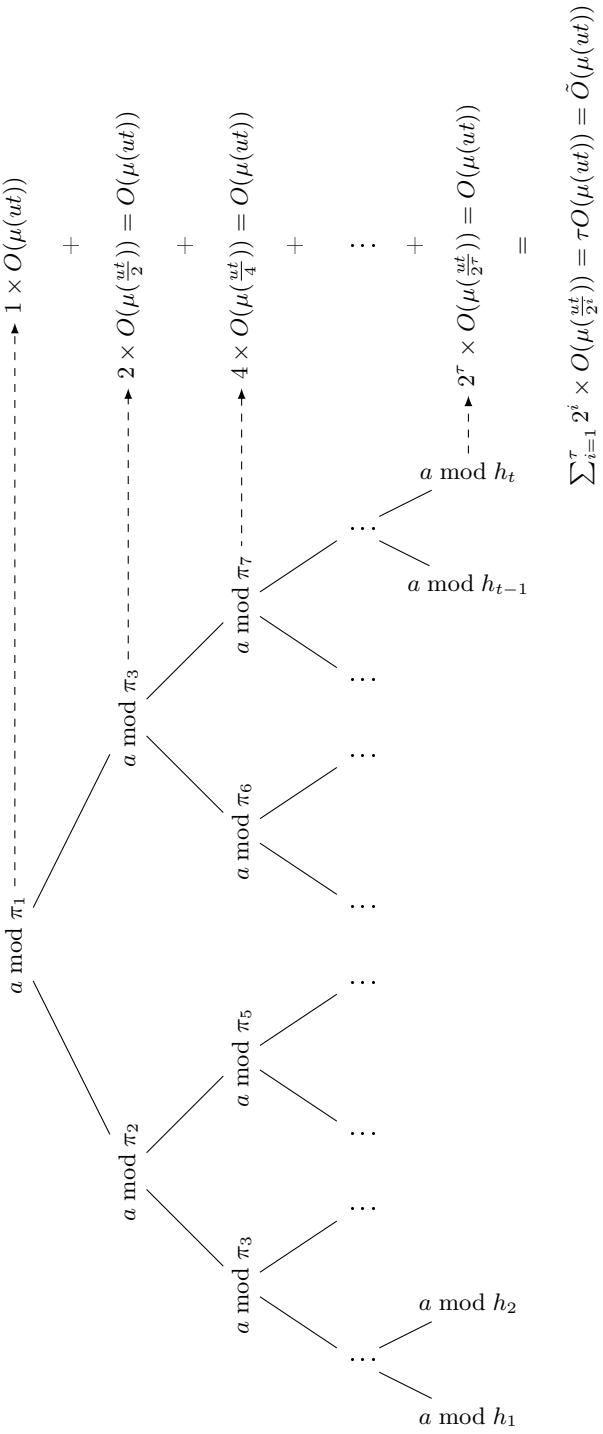

Fig. 4. Modular reduction from product tree (assuming $j = 1$ for simplicity).

Algorithm 4 Division using a product tree

Require: $a \in \mathbb{N}$, π the product tree of Algorithm 3
Ensure: $A[i] = a \bmod \pi_i$ for $i \in \{1, \ldots, 2t-1\}$

```
1: A ← array of size t
2: function modTree(i)
3:     if i < 2t then
4:         A[i] ← A[⌊i/2⌋] mod π_i
5:         modTree(2i)
6:         modTree(2i + 1)
7: A[1] ← a mod π_1
8: modTree(2)
9: modTree(3)
```