Demonstrating data possession and uncheatable data transfer

Décio Luiz Gazzoni Filho, Paulo Sérgio Licciardi Messeder Barreto

[1]Departamento de Engenharia de Computação e Sistemas Digitais (PCS)
Escola Politécnica
Universidade de São Paulo
Av. Prof. Luciano Gualberto trav. 3 n. 158
05508-900 – São Paulo – SP

`decio@decpp.net,pbarreto@larc.usp.br`

Abstract. *We observe that a certain RSA-based secure hash function is* homomorphic. *We describe a protocol based on this hash function which prevents 'cheating' in a data transfer transaction, while placing little burden on the trusted third party that oversees the protocol. We also describe a cryptographic protocol based on similar principles, through which a prover can demonstrate possession of an arbitrary set of data known to the verifier. The verifier isn't required to have this data at hand during the protocol execution, but rather only a small hash of it. The protocol is also provably as secure as integer factoring.*

1. Introduction

The combined computational power of Internet-connected PCs has long been known and applied to the solution of otherwise insurmountable computational problems, spawning the field of distributed computation. Around the turn of the century, peer-to-peer file sharing networks started to appear, showcasing the first application of harnessing the storage space and idle bandwidth of Internet-connected PCs. Recently other applications of this concept have appeared, such as distributed data storage networks with a focus on anonymity [1] and reliable data backup [5]. Entertainment companies are looking to build similar networks, as online hosting and bandwidth costs skyrocket with the introduction of high-quality high-definition content[1]. Content distribution networks relying on users' donated bandwidth and storage space will slash costs dramatically compared to traditional setups.

However, this raises many concerns over the honesty of network users. More explicitly, it is desirable to know, in a distributed data store network, whether users are actually storing files they were assigned to store; and in a content distribution network where users are rewarded for donating their idle bandwidth and charged for using other users' bandwidth, whether data transfers took place correctly and bandwidth credits for the transaction can be exchanged. We cite some scenarios in Sections 3 and 4 where it would make sense for a user to break network rules in exchange for some form of personal gain (or plain vandalism). Hence it is desirable to keep tabs on the honesty of users, which is the purpose of the protocols described in this paper.

It is interesting to note that a personal computer has three main resources which may be exploited by a distributed network: processing time, network bandwidth and storage. In each case a user may 'cheat' and not dedicate the resources as promised. For

[1]For instance, the recently introduced Blu-ray disc format for high-definition video provides storage of up to 50 GB initially and 200 GB in the future[9].

the first, efficient techniques are known for certain classes of problems which detect with arbitrarily high accuracy any attempt to shortcut computations [4]. In this paper, we provide limited (yet still useful) ways to prevent cheating when the latter two resources are involved.

This paper is organized as follows: Section 2 describes an homomorphic hashing scheme, with applications to network coding and more directly to the protocol of Section 3 which can resolve disputes concerning the integrity of data transfers in the face of cheating users. In Section 4 we present a protocol to determine whether a user has a certain chunk of data in storage, or has deleted it and is falsely claiming to still possess it. In Section 5 we analyze the performance of our proposed protocols, and provide some guidance regarding parameter choice. We conclude the paper in Section 6.

1.1. Related work

The first display of a secure homomorphic hash function is due to Krohn, Freedman and Mazires [6]. Their function is mostly satisfactory, despite performance issues (as is the case with the function of Section 2). Our proposal has one main advantage: the same parameter set can be applied to differently-sized messages. Just as a matter of choice, it is interesting to know that a second construction exists, based on a different hard problem (namely factoring), even if it sports the same characteristics and performance[2].

The protocols of Sections 3 and 4 are related to the work of Golle, Jarecki and Mironov [3]. The protocol of Section 3 seeks to solve a different problem than what Golle, Jarecki and Mironov consider. Speaking in terms of uploaders and downloaders, their work considers a trusted uploader and untrusted downloaders, while we consider the case where neither uploaders nor downloaders are trusted; if the protocol fails, it is possible to determine whether the failure was on the uploaders' or the downloaders' end. As for the protocol of Section 4, it has similar goals as Golle, Jarecki and Mironov's proposal. Our protocol has one main advantage: public keys in their protocol are as large as the data being protected, while our protocol's public key is just an RSA modulus. Also, we argue that our protocol is slightly more flexible, as it does not fix the message size for a given parameter set, and is arguably simpler and more elegant. On the other hand, their proposal may have better performance when elliptic curve groups are employed. Regardless of feature set and performance differences, we argue that having a second construction with similar properties, but based on a different hard problem, is good for diversity.

2. An RSA-based homomorphic hash function

A function H is homomorphic if, given two operations $+$ and $\times$, we have

$$H(d + d') = H(d) \times H(d').$$

An homomorphic hash function is, simply put, a hash function that is homomorphic. In many cases it is undesirable that a hash function be homomorphic, and most known constructs of this type are weak. However, it is possible to build strong homomorphic hash functions based on public-key primitives, so long as the secret parameters are

[2]Actually, if elliptic curves are used instead of the group $(\mathbb{Z}/p\mathbb{Z})^*$, one expects improved performance.

not disclosed; up until now, the only known example was the work of Krohn, Freedman and Mazires [6], based on discrete logarithms. We now describe a different homomorphic hash function, based on principles similar to RSA, and which is slightly more flexible than Krohn-Freedman-Mazires's function. This function isn't novel; see e.g. [10]. However, we believe nobody has yet called to attention its homomorphic property.

Let n be an RSA modulus (i.e. $n = pq$ where p and q are primes), and let $\phi(n) = (p-1)(q-1)$ be the order of $(\mathbb{Z}/n\mathbb{Z})^*$. The public data for the hash function is n and a randomly-chosen integer b. To hash a chunk of data d of arbitrary size, one just computes

$$H(d) = b^d \bmod n. \tag{1}$$

This scheme is homomorphic under integer addition: $H(d + d') \equiv b^{d+d'} \equiv b^d b^{d'} \equiv H(d)H(d') \pmod{n}$.

Finding a collision for this hash involves finding messages d, d' such that $H(d) = H(d')$. That is, $b^d \equiv b^{d'} \pmod{n}$, so that $b^{d-d'} \equiv 1 \pmod{n}$. By a theorem of Fermat and Euler, $d - d'$ must be a multiple of $\phi(n)$. Hence, the problem is reduced to finding congruent integers modulo $\phi(n)$. This is trivial if $\phi(n)$ (or the factorization of n) is known, but believed to be difficult otherwise.

The main advantage of our function over Krohn-Freedman-Mazires' function is that the message being hashed can be of arbitrary size, whereas each instance of Krohn-Freedman-Mazires' function fixes the message size.

3. Uncheatable data transfer

We consider a 'pay-for-bandwidth' content distribution network. Suppose that Alice has previously downloaded a piece of data which Bob new wants. Bob would then request this data from Alice in exchange for a virtual (perhaps even real) currency. This could be an attractive model for the distribution of large content such as DVDs and high definition video – users could exchange their idle bandwidth for free content and relieve the network operators from bandwidth charges.

It is natural to assume that some users will attempt to illicitly obtain or avoid parting with bandwidth credits. Two scenarios easily come to mind:

1. Alice has lots of idle bandwidth but not enough interesting data to fully employ this idle bandwidth. Hence she may claim to possess a piece of highly desired data, but when it is requested, she sends unrelated or random data instead.
2. Bob doesn't want to part with his bandwidth credits, so upon successful receipt of data, he will nonetheless claim that the data was corrupt and refuse to hand over the credits.

We remark that the first scenario could be avoided by the use of our novel data possession proving techniques, described in Section 4. However, even users known to possess the correct data may send incorrect data out of malice, hence we might as well directly address the problem.

In order to counter this threat, an hypothetical protocol designer might propose the following schemes:

1. A central server keeps track of hashes for each piece of data in the network. Upon data delivery, Bob hashes the received data, which is compared with the server-kept hash value. If the values don't match, the transfer was corrupted and no credits are exchanged. This scheme is secure against scenario 1 but not 2, since the server has to trust Bob to correctly hash the data.
2. Alice encrypts the data before sending to Bob. Upon data delivery, Bob hashes the received data, which is compared with Alice's hash of the encrypted data. If it matches, Alice supplies the decryption key to Bob. This scheme is secure against scenario 2 but not 1, since Alice might have encrypted random or unrequested data. This could be combined with the first protocol by hashing the decrypted data, however the scheme would again be insecure against scenario 2.
3. The second protocol could be extended by requiring Alice to provide the decryption key to the central server. This scheme is secure against both scenarios, however the server must keep a valid copy of the data to verify the hash of Alice's encrypted data.

Protocol 3 is fairly secure, but rather burdensome for the server, requiring it to store a valid copy of every piece of data being traded in the network. We now show how homomorphic hashing and stream ciphers can be employed to remove this requirement.

Let d be the data of interest and $f(k)$ be a stream cipher; that is, it produces a bitstring s of arbitrary length which depends on k. For our purposes, this bitstring is truncated to match the size of d. Typically d and s would be combined by the XOR operation; however, since we have a hash H which is homomorphic under integer addition, we will interpret both d and s as binary integers, and define $c = s + d$ as the ciphertext, where $+$ is integer addition.

The central server must generate an RSA modulus $n = pq$, where p, q are large primes and $\phi(n) = (p-1)(q-1)$. The modulus n is public, but $p, q, \phi(n)$ are kept secret. Each piece of data d is hashed as $h(d) = d \bmod \phi(n)$ and this value is stored by the server. Alice sends data to Bob according to the following protocol:
Protocol 3.1 (Data transfer)**.**

1. Alice chooses a key k at random for the stream cipher $f(k)$, and computes the bitstring s and the ciphertext $c = s + d$ using this key;
2. Alice sends s and $r_A = H(c)$ to Bob;
3. Bob locally computes $r_B = H(c)$;
4. If $r_A = r_B$, Bob requests the decryption key k from Alice;
5. Bob decrypts the data by computing the bitstring s and $d = c - s$, and checks the integrity of d by conventional means (e.g. a traditional hash function or cyclic redundancy check).

Whenever a dispute arises concerning the integrity of a data transfer, the following protocol is executed:
Protocol 3.2 (Data transfer verification)**.**

1. Alice or Bob supply k and r to the central server;
2. The server computes the bitstring s and its hash $H(s)$;
3. The server computes $r_S = H(c) = H(s)h(d)$;
4. If $r = r_S$, the server declares that Alice correctly sent the data.

The only step that may require clarification is Step 3 of Protocol 3.2. Recall that $\phi(n)$ is the order of $(\mathbb{Z}/n\mathbb{Z})^*$ (the group over which arithmetic takes place), and by definition $h(d) \equiv d \pmod{\phi(n)}$, so that $b^{h(d)} \equiv b^d \pmod{n}$. Since $H(d) = b^d \bmod n$, we have that $H(c) = H(s + d) = H(s)H(d) = H(s)h(d)$ using the homomorphic property of H.

We remark that the server can shortcut the computation of $H(s)$ by computing $H(s \bmod \phi(n))$ instead; nevertheless, we posit that performance is not an issue since this protocol will be executed very infrequently, as a rational cheater would steer away from networks implementing this protocol.

The attractiveness of this protocol relies on the fact that the server can hash the encrypted version of d without having a copy of d itself, but only its hash $h(d)$. It is otherwise functionally equivalent to the third protocol proposed in the beginning of this section. Moreover, the protocol is secure, since our homomorphic hash has the desired security properties of a conventional hash, as long as the secret RSA parameters are not revealed. Only the central server can compromise the hash function's security, but this isn't a problem since the central server doesn't produce any hashes under the protocol.

An issue that must not be overlooked is the commitment of r_A and r_B in Protocol 3.1. If this value is not committed, the cheater (whether it be Alice or Bob) may provide a different value of r to the central server during execution of Protocol 3.2, and the server has no way of telling which of the two is being honest. For instance, Alice may choose a key k' and compute the corresponding bitstring s', producing a different hash $r'_A = H(s' + d)$ which is consistent with the key k'. The protocol would execute successfully and the dispute would be incorrectly resolved in favor of Alice. This commitment might be in the form of sending r_A to the central server as well in Step 2 of Protocol 3.1, or by having Alice sign r_A with her public key before sending it to Bob, and then having Bob sign it as well before Alice supplies the decryption key.

4. Demonstrating data possession

In the previous section, one of the contemplated cheating scenarios concerned a user with idle bandwidth but no files of interest, which might be tempted to claim possession of data which he doesn't have, in order to maximize his bandwidth output. This is prevented by the use of our data possession proving protocol.

Another particularly useful application of this protocol is in distributed data store systems. Consider for instance the Freenet network [1], or a distributed backup system such as Allmydata (formerly HiveCache) [5]. Such systems rely not only on the idle bandwidth of users, but free hard drive storage as well. There is extra reason to cheat under such a scheme, as storage space has real monetary cost attached to it (unlike idle bandwidth). Moreover, in a distributed backup system, a user that donates x gigabytes of space towards storing other users' backup data is usually given back x gigabytes of space somewhere in the network for his own backup, so that a user may cheat in order to obtain better redundancy for his own data. Finally, users will only place trust on such a system as long as the storage can be consistently relied upon, which is difficult in the face of massive cheating, even if network coding is employed. It is then desirable to verify that users are actually storing what they claim to be.

To that end, we consider a setup involving an RSA modulus $n = pq$, with

$p, q, \phi(n)$ as in Section 3. Suppose Bob has uploaded a piece of arbitrary data d to Alice, which has agreed to store it in her computer's hard drive. Bob may even delete the data d, so long as he keeps a hash $h(d) = d \bmod \phi(n)$. Later on, Bob wishes to ascertain that Alice hasn't deleted d yet, and so executes the following protocol:
Protocol 4.1 (Demonstrating data possession)**.**

1. Bob chooses an integer $1 < b < n-1$ at random, and sends it to Alice;
2. Alice computes $r = b^d \bmod n$ and sends it to Bob;
3. Bob computes $r' = b^{h(d)} \bmod n$;
4. If $r = r'$, Bob is convinced that Alice hasn't deleted d.

The correctness of Protocol 4.1 follows from the same argument for the correctness of Protocol 3.2.

Given the similarities with the homomorphic hash function of Section 2, it isn't surprising that the homomorphic property is valid for Protocol 4.1. That is, if $c = s + d$ ($+$ being integer addition), then $r_c \equiv b^c \equiv b^{s+d} \equiv b^s b^d \equiv r_s r_d \pmod{n}$. This is particularly advantageous when data is erasure encoded using a parity check code, such as tornado codes [7], online codes [8], etc., as long as these codes are modified to use integer addition instead of XOR to combine basic blocks into encoded blocks. Instead of storing the hash $h(d)$ for every encoded block that's been uploaded to the network, it is possible to store the hash of basic blocks only, and combine them using the homomorphic property to form the hashes of the encoded blocks being sought.

4.1. Security and limitations

The choice of base b is crucial for security and performance. Clearly b should be prime; for if b is composite, a cheater may build a factor base. For instance, it is possible to successfully complete the protocol for $b = 6$ without knowing d, as long as the results for $b = 2$ and $b = 3$ are known. If d is sufficiently large and only a small number of bases were used, including composite values, it might make sense to precompute the results of the protocol for many admissible b's. A less obvious possibility is: given a set of primes $p_1, \ldots, p_k$, is it possible that $p_i = \prod_{j \in \mathcal{S}} p_j \pmod{n}$ for some subset $\mathcal{S}$ of $p_1, \ldots, p_k$? Clearly the answer is negative over the integers, due to the Fundamental Theorem Of Arithmetic, but we have no idea whether it is valid modulo n, leaving this as an open question. For performance reasons, it is desirable to employ small bases b, as discussed in Section 5.

The applicability of this protocol is restricted if the same data is stored by multiple network users, a common scenario in current peer-to-peer networks. These users may collude so that only one of them stores the data; that way, each user in a collusion of n users, each with a storage space of g bytes, will be able to claim a storage space of ng bytes. Whenever a request for protocol execution is made to one of the members of the collusion, it is forwarded to the member who is actually storing the data. However, this isn't a weakness of the protocol itself, but rather a statement about the class of protocols with the same purpose: network users can always collude and forward protocol verifications to others. Nevertheless, in a content distribution network employing digital rights management, where content is encrypted and the key is not explicitly given to the user, this protocol may be applicable as each member of the collusion would actually be storing different data (the plaintext may be the same, but encrypted under different keys).

5. Performance

Performance of the protocols presented here is low, mostly due to the requirement of 1-2 modular multiplications per bit. It is an important open problem to find more efficient primitives with the desired properties, hopefully leading to an asymptotic reduction in multiplication count, as in e.g. VSH [2]. Failing that, even a constant factor improvement through the use of elliptic curves would be very welcome[3] We remark that Krohn-Freedman-Mazires' homomorphic hash function can be trivially adopted to use elliptic curve arithmetic, and is a suitable hash function for use with Protocols 3.1 and 3.2 (although key management is necessarily less flexible, since a different set of parameters for Krohn-Freedman-Mazires' function is as long as the data itself, while the hash function of Section 2 requires the storage of a single RSA key, typically 512-1024 bits.) Golle, Jarecki and Mironov's protocol [3] for demonstrating data possession is a suitable replacement for Protocol 4.1 if extra performance is required, and one does not require the slight extra flexibility afforded by our protocol.

Nevertheless, the protocols presented here have some rather unique features which are desirable in certain applications, and it may be that their low performance is acceptable since the alternatives (if any) are even less efficient. Thus, we shall consider the problem of efficiently selecting parameters and implementing our protocols.

The most gain can be made from selecting small key sizes. We shall argue that, for the present, a key size of 512 bits is perfectly sufficient for the applications outlined here (content distribution networks and distributed data stores). Consider what cheaters have to gain from breaking this scheme: they'll avoid using some bandwidth or hard drive storage space, both of which are fairly cheap resources compared to those involved in cracking 512-bit or larger RSA keys. In fact, the authors are not aware of a single factorization of a 512-bit or larger RSA modulus performed by a single individual with a small network of PCs. Hence, unless key management is done very poorly (using the same key for many gigabytes or terabytes of data, or perhaps even a master key for the whole network), it makes no economic sense to break these keys; cheaters would be better off buying larger hard drives or a faster Internet connection instead of the computational resources required for key-breaking.

A second improvement is in the choice of base b, either for the hash function of Section 2 or Protocol 4.1. Consider one of the simplest exponentiation algorithms, which is left-right exponentiation; the algorithm loops over the number of bits of the exponent, performing a squaring and, in case the current exponent bit is 1, a multiplication by b. If b is small, a multiplication can be replaced by an addition chain (in exactly the same way as an exponentiation is replaced by a multiplication chain). In the simplest case, $b = 2$, a multiplication is replaced by a single addition. However, one must heed the warnings of Section 4.1 regarding the unknown security of a scheme using small bases. Also, if somewhat larger bases are used, the cost of multiplications is no longer negligible compared to the cost of squarings, and the fairly large amount of multiplications involved in a left-right algorithm must be weighed against more efficient exponentiation algorithms, such as k-ary or sliding window exponentiation, which require full-precision multiplications

[3]Note that this is a constant factor improvement only if key sizes are fixed; for varying key sizes, obviously elliptic curves are an asymptotic improvement over RSA and $(\mathbb{Z}/p\mathbb{Z})^*$ discrete logarithm based schemes.

	Client/Prover			Server/Verifier		
Data size $\setminus$ Key size	512 bits	768 bits	1024 bits	512 bits	768 bits	1024 bits
4 KB	21.7 ms	41.6 ms	68.4 ms	1.10 ms	2.84 ms	5.96 ms
64 KB	333 ms	644 ms	1.05 s	1.46 ms	3.28 ms	6.44 ms
1024 KB	5.20 s	10.0 s	16.4 s	8.56 ms	12.5 ms	17.9 ms

Table 1. Timings on a PowerPC G4 1.42 GHz processor.

	Client/Prover			Server/Verifier		
Data size $\setminus$ Key size	512 bits	768 bits	1024 bits	512 bits	768 bits	1024 bits
4 KB	9.96 ms	10.8 ms	28.3 ms	0.402 ms	0.684 ms	2.73 ms
64 KB	152 ms	164 ms	436 s	0.670 ms	0.979 ms	3.22 ms
1024 KB	2.41 s	2.61 s	6.99 s	5.53 ms	6.35 ms	9.80 ms

Table 2. Timings on a Pentium 4 3.0 GHz processor.

but not as many of them.

To gauge the real-world performance of our protocols, we implemented the hash function of Section 2 using the GNU MP library. There were two distinct code paths:

- the client/prover code path, which is just a straightforward implementation of Eq. (1);
- the server/verifier code path, which requires knowledge of the prime factors of the RSA modulus n, implementing the Chinese Remainder Theorem method for faster computation modulo n and the shortcut discussed in Section 3, which consists of reducing the exponent modulo $\phi(n)$.

We used key sizes of 512, 768 and 1024 bits, and data sizes of 4 KB, 64 KB and 1024 KB. Bases were chosen at random with the same bit size as the respective keys, and exponentiation was performed using GMP's `mpz_powm` function, which implements sliding-window exponentiation. Each test was re-run five times with the same random data and the results were averaged. Two processors were benchmarked: a PowerPC G4 1.42 GHz (Table 1) and an Intel Pentium 4 3.0 GHz (Table 2), both of them 32-bit processors. The benchmark was linked against GMP version 4.1.4 for the G4 processor and 4.2.0 for the Pentium 4 processor.

Results for the client/prover case are clear, with a processing rate of barely more than 400 KB/s for a midrange processor and less than 200 KB/s for a low end processor, using the lowest security level. It is not uncommon to see Internet connectivity speeds surpassing these values, rendering the protocols unsuitable for real time processing of data. It is hard to argue for even smaller keys; perhaps 384 bits if keys are temporary, and even then performance would barely double. System designers must work around this limitation by avoiding protocol executions unless there's a proven need for them, e.g. when there have been repeated cheating reports about a specific user.

For the server/verifier case, performance is reasonable and scales better. For instance, at the lowest security level, a Pentium 4 3.0 GHz serving users which execute the protocol once a minute to check on 1024 KB of data could support in excess of 10,000 users.

6. Conclusions

We presented new protocols to aid the task of eliminating cheaters/freeriders from content distribution and distributed data store networks. Although they are very flexible and simple to implement, low performance may prevent their widespread adoption. Hence we urge further research in this area, specifically in trying to adapt these protocols to use elliptic curve cryptographic, which may provide the required performance boost to render these techniques mostly practical.

References

[1] Ian Clarke. Freenet – the free network project. `http://freenetproject.org`.

[2] Scott Contini, Arjen K. Lenstra, and Ron Steinfeld. VSH, an efficient and provable collision resistant hash function. `http://eprint.iacr.org/2005/193`, 2005.

[3] Philippe Golle, Stanislaw Jarecki, and Ilya Mironov. Cryptographic primitives enforcing communication and storage complexity. In *Financial Cryptography*, volume 2357 of *Lecture Notes in Computer Science*, 2002.

[4] Philippe Golle and Ilya Mironov. Uncheatable distributed computations. *Lecture Notes in Computer Science*, 2020:425+, 2001.

[5] Allmydata Inc. Allmydata. `http://allmydata.com`.

[6] Maxwell Krohn, Michael Freedman, and David Mazières. On-the-fly verification of rateless erasure codes for efficient content distribution. In *Proc. IEEE Symposium on Security and Privacy*, pages 226–240, Oakland, CA, 2004.

[7] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, Daniel A. Spielman, and Volker Stemann. Practical loss-resilient codes. In *Proc. 29th Symposyum on Theory of Computing*, pages 150–159, 1997.

[8] Petar Maymounkov. Online codes. Technical Report TR2002-833, New York University, October 2002.

[9] Blu ray Disc Association. Blu-ray disc format. `http://www.blu-raydisc.com/`.

[10] Adi Shamir and Yael Tauman. Improved online/offline signature schemes. In *Advances in Cryptology – CRYPTO 01*, volume 2139 of *Lecture Notes in Computer Science*, pages 355–357, 2001.