On-the-Fly Verification of Erasure-Encoded File Transfers (Extended Abstract)

Max Krohn and Michael J. Freedman
New York University
{max,mfreed}@cs.nyu.edu

Abstract

The quality of peer-to-peer content distribution can suffer from the malicious behavior of participants that corrupt or mislabel content. While systems using simple block-by-block downloading can verify blocks using traditional cryptographic signatures, these same techniques may not be applied to more elegant systems that rely on erasure codes for efficient multi-source download. This paper presents a practical scheme, based on homomorphic hashing, that enables a downloader to perform on-the-fly verification of erasure-encoded blocks.

1 Introduction

The Problem. Many users of peer-to-peer content-distribution networks have had the experience of waiting for a large download to complete, only to discover that the endeavor was doomed from the very first byte: the file in question was mislabeled. Some content publishers accidentally err in naming their files. Others maliciously distribute incorrect content to either promote their own preferences, or to detract from the experience of "honest" users of the system.

This paper takes up the problem of on-the-fly content verification, both from the perspective of the publisher and the downloader. Our goal is to allow a downloader to detect unauthenticated content almost immediately.

Consider content-distribution networks composed of three types of actors: publishers, mirrors, and downloaders. Publishers introduce files into the network and cryptographically sign files they publish. Unlike mirrors, they are well-known or trusted by the downloaders. If a downloader has access to a publisher's public key, she can verify that a file she downloaded from a mirror is the very same file that the publisher originally published. In traditional, centralized content distribution networks, mirrors are often honest, so a downloader will almost always retrieve the file that the publisher intended. In a peer-to-peer content-distribution network, mirrors may be frequently malicious.

A traditional publisher signs a file by cryptographically hashing it with a hash function such as SHA-1, and then signing the hash with a public-key signature system such as RSA. Thus, a downloader must wait for a download to complete before verifying it. One can imagine naïve improvements on this scheme that allow a downloader to verify a file on-the-fly. A publisher might break a large file into a sequence of much-smaller chunks and sign a concatenation of the hashes of those chunks. In this scheme, when a downloader encounters a chunk boundary, she hashes the chunk and verifies that it matches the publisher's hash. If not, then the downloader terminates the download and finds another, hopefully more honest mirror. Chunks that hashed correctly earlier in the download clearly need not be thrown away.

Motivation. A recent trend toward erasure encoding of large file transfers in peer-to-peer networks complicates the picture [2, 5]. A downloader can download the same file from multiple mirrors concurrently. By using erasure codes, the downloader need not coordinate which mirrors send which parts of the file. Rather, all mirrors generate a random stream of encoded file blocks, and once the receiver has received a sufficient amount of these blocks, she can reconstruct the original file. A shallow analysis suggests that erasure encoding and the file-chunk-hashing scheme mentioned above are incompatible. The publisher cannot possibly hash all possible block encodings that the mirror might produce, nor can the downloader establish any connection between the hashes of encoded blocks and the encoding of hashed blocks.

This paper proposes a scheme that enables a downloader to perform on-the-fly verification of erasure-encoded blocks. We construct a discrete-log-based homomorphic hash function for such a purpose, and describe how to make this scheme efficient by probabilistic batch verification. We believe that the authenticated distribution of publishers' public keys is an orthogonal problem, and hence we do not consider it here.

2 Brief Review of Erasure Codes

Consider a large file $\mathbf{F}$, comprised of n uniformly-sized blocks, $B_1 \,||\, B_2 \,||\, \ldots \,||\, B_n$, known as *message blocks*. Most efficient erasure-encoding schemes [3, 4] use the bitwise XOR operation ($\oplus$) over messages blocks to produce *checkblocks*. For instance, a checkblock C_1 might be defined as $C_1 = B_2 \oplus B_{10}$. A checkblock's *degree* is defined to be the number of message blocks that compose it. In this example, the degree of C is 2. Thus, message blocks can be thought of as checkblocks with degree 1. An encoding of a file consists of a sequence of checkblocks.[1] The properties of the particular code determine how the check blocks are constructed. "Rateless" erasure codes — those that generate an arbitrary-length sequence of blocks — are well-suited for multi-sourced download in content-distribution networks.

[1] The encoding also includes index information, either implicit or explicit, so that the receiver knows which message blocks comprise the checkblocks.

A downloader decodes a file by iteratively discovering message blocks. If she has received C_1 as above, and $C_2 = B_2$, the she can recover $B_{10} = C_1 \oplus B_2$. The properties of a given erasure code ensure that downloader can reconstruct the original file $\mathbf{F}$ with high probability after receiving, *e.g.*, $(1+\epsilon)n$ checkblocks for small ϵ, depending on the code.

The operation used to form checkblocks need not be the typical bitwise XOR, however. Rather, it need only be fast to compute, and easy to invert.

3 Our Solution: Homomorphic Hashing

This section presents a homomorphic collision-resistant hash function (CRHF) that enables a downloader to verify encoded blocks retrieved from mirrors.

3.1 Mathematical Primitives

Each publisher picks hash parameters, which can be used for multiple files. Vector exponentiation is defined component-wise, *i.e.*, if $\mathbf{r} = (r_1 \ \cdots \ r_m)$, then $g^{\mathbf{r}} = (g^{r_1} \ \cdots \ g^{r_m})$.

name	description	*e.g.*
λ	discrete log security parameter	1024
p	random prime, $\|p\| = \lambda$	$\|p\| = 1024$
q	random prime, $q \| (p-1)$	$\|q\| = 257$
β	block size in bits	65536
m	$= \lceil \beta / \|q-1\| \rceil$	256
g	random element in $\mathbb{Z}_p^*$, order q	
$\mathbf{r}$	random row vector in $\mathbb{Z}_q^m$	
$\mathbf{g}$	$= g^{\mathbf{r}}$	
G	the group, given by $(p, q, \mathbf{g})$	
d	average degree of checkblocks	10

Unless stated otherwise, all addition is taken over $\mathbb{Z}_q$, and all multiplication (and exponentiation) is taken over $\mathbb{Z}_p^*$.

Encoding. To encode a file $\mathbf{F}$, consider it as an $m \times n$ matrix in $\mathbb{Z}_q^{m \times n}$, where $n = \lceil |\mathbf{F}| / \beta \rceil$. Each file block $\mathbf{b_j}$ then consists of columns of the matrix, or rather, m $|q|$-bit sub-blocks. We write $\mathbf{b_j} = (b_{1,j}, \ldots, b_{m,j})$, thus:

$$\mathbf{F} = (\mathbf{b}_1 \ \cdots \ \mathbf{b_n}) = \left\{ \{b_{i,j}\}_{i=1}^{m} \right\}_{j=1}^{n}$$

Addition of blocks is defined as component-wise addition of their corresponding columns in the matrix. That is, to combine the i^{th} and j^{th} blocks of the file, compute:

$$\mathbf{b_i} + \mathbf{b_j} = (b_{1,i} + b_{1,j}, \ldots, b_{m,i} + b_{m,j}) \bmod q$$

Now, proceed with erasure encoding as normal, using vector addition over $\mathbb{Z}_q$ instead of the $\oplus$ operator.

Hash Generation. To hash a file, we use a collision-resistant hash function (CRHF), secure under the discrete-log assumption. Recall that a CRHF is informally defined as a function for which finding any two inputs that yield the same hash-value is difficult.

For an arbitrary file block $\mathbf{b_j}$, define its hash in G:

$$h_G(\mathbf{b_j}) = \prod_{i=1}^{m} g_i^{b_{i,j}} \bmod p \tag{1}$$

Define the hash of a complete file $\mathbf{F}$ as a $1 \times n$ row-vector whose elements are the hashes of its constituent blocks:

$$H_G(\mathbf{F}) = (h_G(\mathbf{b_1}) \ \cdots \ h_G(\mathbf{b_n})) \tag{2}$$

Since the publisher generated $\mathbf{g} = g^{\mathbf{r}}$ in the first place, she can compute $H_G(\mathbf{F})$ directly and efficiently:

$$H_G(\mathbf{F}) = g^{\mathbf{rF}} \tag{3}$$

Note that $|H_G(\mathbf{F})| = |\mathbf{F}|/m$. Thus, H_G is length-reducing, but unlike more conventional hashes (such as SHA-1), does not have a fixed output size.

Hash Verification. If a downloader knows $H_G(\mathbf{F})$, he can verify any checkblock of the form

$$\mathbf{c} = \sum_{k \in S} \mathbf{b_k} \text{ for any } S \subset \{1, .., n\}$$

by verifying that

$$h_G(\mathbf{c}) = \prod_{k \in S} h_G(\mathbf{b_k}) \tag{4}$$

Note that H_G functions here as a *homomorphic hash function*. We are using the fact that addition distributes over multiplication in $\mathbb{Z}_p$ to verify checkblocks.

Decoding. A downloader can reconstruct the file $\mathbf{F}$ after receiving a sufficient number of checkblocks, $n' = (1+\epsilon)n$, for small ϵ. Call these checkblocks $\mathbf{c_1}, \ldots, \mathbf{c_{n'}}$. To decode, we use scalar multiplication of a block $\mathbf{b_j}$ by $x \in \mathbb{Z}_q$, defined as normal:

$$x\mathbf{b_j} = (xb_{1,j}, \ldots, xb_{m,j})$$

The decoding algorithm proceeds as normal, but with the use of additive inverses over the m-vectors. Given $\mathbf{c} = \mathbf{b_j} + \mathbf{b_k}$ and $\mathbf{b_j}$, $\mathbf{b_k} = \mathbf{c} - \mathbf{b_j}$.

3.2 Content-Distribution Protocol

Publishers, mirrors, and downloaders behave as follows.

Publishers. When publishing a file $\mathbf{F}$, a publisher P should split it into blocks and sub-blocks as mentioned above. She then should compute $H_G(\mathbf{F})$, and publish $\mathbf{F}$ and $((G, H_G(\mathbf{F})), \sigma_P(G, H_G(\mathbf{F})))$ where σ is a public-key signature scheme that accepts arbitrary-length inputs. Publishers should use erasure codes to distribute large files $\mathbf{F}$, but should distribute $((G, H_G(\mathbf{F})), \sigma_P(G, H_G(\mathbf{F})))$ as an unencoded, single-source download. Finally, we note publishers must be careful never to disclose their $\mathbf{r}$ vectors. So doing would enable malicious parties to compute arbitrarily many hash-collisions for their function H_G.[2]

[2] In the given protocol, the hashes of files might be large and hence their transmission might be susceptible to bandwidth-wasting attacks. A slightly more complex system allows the signatures themselves to be treated as normal files; iteratively invoking the protocol i times yields hashes $1/2^{8i}$ the size of the original file. Further details are beyond the scope of this paper.

operation	$\mathsf{MultCost}(q)$ per block	*e.g.*	$\mathsf{MultCost}(p)$ per block	*e.g.*	approx. CPU time
Hash Generation	m	256	$\lvert q\rvert/(2n)$	0	256 μsecs
Naïve Verification	0	0	$m(\lvert q\rvert/2+1)+d$	32788	131 msec
Fast Verification	m	256	$d+l/2+\lceil m(\lvert q\rvert/2+1)+l\rceil/t$	294	1.3 msecs

Figure 1: Computational costs required by our different algorithms. Note that $n \gg |q|$, and we approximate $\mathsf{MultCost}(p) \approx 4$ μsecs and $\mathsf{MultCost}(q) \approx .5$ μsecs on a 1.75GHz Athlon XP. Also, we do not consider the cost of additions.

Mirrors. An honest mirror M shares a file $\mathbf{F}$ from a publisher P first by downloading the signature $((G, H_G(\mathbf{F})), \sigma_P(G, H_G(\mathbf{F})))$ from P or another mirror. M validates this signature using P's public key. Assuming that the signature validates, M then starts a (potentially multi-sourced) erasure-encoded transfer of the file $\mathbf{F}$. If ever any checkblock $\mathbf{c}$ yields a hash $h_G(\mathbf{c})$ that fails to equal the hash given the appropriate linear combination of elements of $H_G(\mathbf{F})$, then the mirror will disconnect from the corrupted source. When downloaders or other mirrors request the file $\mathbf{F}$ from M, M will transmit the same $((G, H_G(\mathbf{F})), \sigma_P(G, H_G(\mathbf{F})))$ that it received from its source. When transferring file $\mathbf{F}$, however, it will formulate its own erasure-encoding of the file. Note that mirrors need never generate $H_G(\mathbf{F})$.

Downloaders. Downloaders behave exactly like mirrors, but clearly do not serve files to other mirrors or downloaders.

3.3 Efficiency Improvements

Our mathematical primitives, as presented above, are rather inefficient. Component-wise addition over $\mathbb{Z}_p^*$ is slower than the bitwise XOR operation. More important, our hash function H_G is order of magnitudes slower than a more conventional hash function like SHA-1. Our most important goal here is to improve the performance of a downloader or mirror verifying the hash of a file $H_G(\mathbf{F})$. The bare bones primitives above imply that a client must recompute $H_G(\mathbf{F})$ and compare it with the $H_G(\mathbf{F})$ signed by the publisher, but without knowing $\mathbf{r}$.

A technique suggested by Bellare, Garay, and Rabin [1] is used to improve verification performance. Instead of verifying each checkblock $\mathbf{c}_i$ exactly and on the fly, we verify them probabilistically and in batches. Please see Appendix A for a fuller description of the batching protocol.

4 Analysis

In this section, we briefly sketch the running time and security analysis of our homomorphic hashing scheme; please see Appendix A for more in-depth analysis and proofs of security.

4.1 Running Time Analysis

In analyzing the running time of our algorithms, we count the number of multiplications over $\mathbb{Z}_p^*$ and $\mathbb{Z}_q$ needed. In our analysis, we denote $\mathsf{MultCost}(p)$ as the cost of multiplication in $\mathbb{Z}_p^*$, and $\mathsf{MultCost}(q)$ as the cost of multiplication in $\mathbb{Z}_q$. Computing arbitrary y^x in $\mathbb{Z}_p^*$ has an expected cost of $|x|\,\mathsf{MultCost}(p)/2$. Figure 3.1 lists the running time of the scheme's formulae.

Hash Generation. Publishers first precompute a table g^{2^z} for all z such that $1 \leq 2^z < q$. To compute $\mathbf{rF}$ as in Equation 3, mn multiplications are needed in $\mathbb{Z}_q$. Disregarding the one-time precomputation of g^{2^z}, each exponentiation in Equation 3 requires an expected $|q|/2$ multiplications in $\mathbb{Z}_p^*$.

Naïve Hash Verification. Hash verifiers first precompute tables of $g_i^{2^z}$ for all g_i, costing $m|q|\,\mathsf{MultCost}(p)/2$. We disregard this cost in Figure 3.1 since for large files, $n \gg m|q|$. Next, verifiers compute $h_G(\mathbf{c})$ without knowing $\mathbf{r}$ via Equation 1 (total cost: $m(|q|/2+1)\mathsf{MultCost}(p)$). Finally, the right side of Equation 4 necessitates d multiplications in $\mathbb{Z}_p^*$, where d, we recall, is the average degree of a checkblock $\mathbf{c}$.

Fast Hash Verification. Appendix A includes a analysis of the batching verification algorithm following its description.

4.2 Security

An adversary in this system is a mirror who attempts to trick downloaders and other mirrors into downloading and accepting a file different from the one that they requested. If a publisher P publishes a file $\mathbf{F}$, and its complete hash and signature $((G, H_G(\mathbf{F})), \sigma_P(G, H_G(\mathbf{F})))$, an adversary attempts to generate a "forged" file $\mathbf{F}'$. When a downloader or mirror requests the file $\mathbf{F}$, the adversary returns the file $\mathbf{F}'$ and the original signature and hash generated by P. A downloader will continue the transfer as long as the checkblocks generated from $\mathbf{F}'$ continue to match the hashes given by $H_G(\mathbf{F})$.

In Appendix B, we show that the fast hash verification scheme is correct, and use the hardness of the discrete log problem to prove the scheme's security: The probability of a false-negative is negligible in the security parameter.

Note that in this paper, we do not consider the problem of how a publisher might reliably distribute his public key to interested downloaders and mirrors. For simplicity, we assume that downloaders and mirror can securely pair publishers with their associate public keys, perhaps through an out-of-band and trusted channel. Nor do we discuss naming semantics that specify the desired content to the downloaded, although self-certifying pathnames [6] are one possible technique.

5 Conclusions

Current peer-to-peer content distribution networks, such as the widely popular file-sharing systems, suffer from unverified downloads: A participant may download an entire file,

increasingly in the hundreds of megabytes, before determining that the file is corrupted or mislabeled. Current downloading techniques can use simple cryptographic primitives, such as signatures and hash trees, to authenticate data. However, the introduction of erasure-encoding for more efficient multi-source download invalidates these straight-forward approaches.

This paper considers the problem of on-the-fly verification of erasure-encoded content. We present a hash scheme based on a discrete-log construction that provides useful homomorphic properties for verifying check blocks against authenticated file blocks. However, this basic construction is quite inefficient; therefore, we apply batching techniques for probabilistic verification that yield an efficient online algorithm. It is an open question whether more efficient schemes exist.

Acknowledgments

We wish to thank Petar Maymounkov, David Mazières, and Benny Pinkas for helpful discussions. This research was conducted as part of the IRIS project (`http://project-iris.net/`), supported by the NSF under Cooperative Agreement No. ANI-0225660.

References

[1] Mihir Bellare, Juan Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In Kaisa Nyberg, editor, *Advances in Cryptology—EUROCRYPT 98*, volume 1403 of *Lecture Notes in Computer Science*. Springer-Verlag, May 31–June 4 1998.

[2] John Byers, Michael Luby, and Michael Mitzenmacher. Accessing multiple mirror sites in parallel: Using Tornado codes to speed up downloads. In *Proceedings of IEEE INFOCOM '99*, pages 275–283, New York, NY, 1999.

[3] Michael Luby. LT codes. In *43rd Annual Symposium on Foundations of Computer Science*, Vancouver, Canada, November 2002. IEEE.

[4] Petar Maymounkov. Online codes. Technical Report 2002-833, NYU, November 2002.

[5] Petar Maymounkov and David Maziéres. Rateless codes and big downloads. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, February 2003.

[6] David Mazières and M. Frans Kaashoek. Escaping the evils of centralized control with self-certifying pathnames. In *Proceedings of the 8th ACM SIGOPS European workshop*, Sintra, Portugal, September 1998.

A Batch Verification of Hashes

Instead of verifying each checkblock $\mathbf{c}_j$ exactly and on the fly, we verify them probabilistically and in batches [1]. Each downloader and mirror picks the following parameters:

name	description	*e.g.*
t	batch size	128 blocks
l	security parameter	32

To verify, the downloader collects t checkblocks of the file $\mathbf{F}$, which for convenience, we will call $\mathbf{C} = (\mathbf{c}_1 \cdots \mathbf{c}_t)$. For each $j \in \{1, .., t\}$ the downloader knows $S_j \subset \{1, .., n\}$ such that $\mathbf{c}_j = \sum_{k \in S_i} \mathbf{b_k}$. She then proceeds as follows:

1. Pick $\mathbf{s} = (s_1, \ldots, s_t) \in \left\{\{0,1\}^l\right\}^t$ at random
2. Compute $\mathbf{z} = \mathbf{Cs}$
3. Compute $h_G(\mathbf{c}_j) = \prod_{k \in S_j} h_G(\mathbf{b_k})$ for all $j \in \{1, .., t\}$.
4. Compute $y' = \prod_{i=1}^{m} g_i^{z_i}$, and $y = \prod_{j=1}^{t} h_G(\mathbf{c}_j)^{s_j}$
5. Verify that $y' \equiv y \bmod p$

In Step 2, we recall that $\mathbf{C}$ is a $m \times t$ matrix, and hence the matrix multiplication costs $mt\,\mathsf{MultCost}(q)$. We determine $h_G(\mathbf{c}_j)$ in Step 3 with d multiplications over $\mathbb{Z}_p^*$, at a total cost of $td\,\mathsf{MultCost}(p)$. In Step 4, we use brute force to compute y' at a cost of $m(|q|/2+1)\,\mathsf{MultCost}(p)$. However, we compute y efficiently with the FastMult algorithm [1], at a total cost of $(l + tl/2)\,\mathsf{MultCost}(p)$.[3] Summing these computations together yields the value given in Figure 3.1.

B Security analysis

B.1 Correctness of Batched Verification

To prove the correctness of batched verification in the semi-honest model, consider an arbitrary pair $(\mathbf{C}, H_G(\mathbf{C}))$. We write y and y' computed in Step 4 in terms of the generator g and the random vector $\mathbf{r}$. Recall that $g_i \equiv g^{r_i} \bmod p$. Thus,

$$y' = \prod_{i=1}^{m} g_i^{z_i} = \prod_{i=1}^{m} g^{r_i z_i} = g^{\sum_{i=1}^{m} z_i r_i} = g^{\mathbf{rz}}$$

By the definition of $\mathbf{z}$ from Step 2, we conclude $y' = g^{\mathbf{rCs}}$.

Now we examine the other side of the verification, y. Recalling Equation 1, and rewriting in terms of the common generator g, we have that:

$$h_G(\mathbf{c_j}) = \prod_{i=1}^{m} g^{r_i c_{i,j}} = g^{\sum_{i=1}^{m} r_i c_{i,j}} = g^{\mathbf{r}\mathbf{c}_j}$$

Combining with the computation of y in Step 4:

$$y = \prod_{j=1}^{t} g^{s_j \mathbf{r}\mathbf{c}_j} = g^{\sum_{j=1}^{t} s_j \mathbf{r}\mathbf{c}_j} = g^{\mathbf{rCs}}$$

Thus we have that $y' \equiv y \bmod p$, proving the correctness of the validator.

B.2 Soundness of Batched Verification

We use the hardness of the discrete log problem to prove the scheme's soundness against a malicious adversary. For the purposes of this proof, define a group G as before, but without the pre-generated vector of generators, $\mathbf{g}$. That is, the group G can be described as the elements of $\mathbb{Z}_p$ of order q. Note that all elements of the group G can be written as g^r, for a fixed g and a random $r \in \mathbb{Z}_q$.

[3] FastMult offers no per-block performance improvement for naïve verification, thus we only consider it for fast verification.

Consider a discrete-log hash adversary $\mathcal{A}_G$ who takes as input a random set of generators in G and attempts to find a collision. Define the probability that $\mathcal{A}_G$ succeeds:

$$\mathsf{Adv}_0(\mathcal{A}_G) = \Pr[g \leftarrow G;\ \mathbf{r} \leftarrow \mathbb{Z}_q^m;\ (\mathbf{a}, \mathbf{a}') \leftarrow \mathcal{A}_G(g^{\mathbf{r}}) : g^{\mathbf{ra}} = g^{\mathbf{ra}'} \ \wedge\ \mathbf{a} \neq \mathbf{a}']$$

Assuming the discrete-log problem is hard in G, $\mathsf{Adv}_0(\mathcal{A}_G)$ is negligible in λ, for all PPT adversaries $\mathcal{A}_G$. The proof is canonical, and we do not reiterate it here.

The adversary in our model takes as input a file $\mathbf{F}$ from a samplable non-uniform distribution of files $\mathcal{F}$, and outputs a potentially "forged" erasure encoding. Let us represent checkblocks recipes as (sparse) row bit vectors in $\{0,1\}^n$. Given such a vector $\mathbf{x}$, the corresponding checkblock is then the vector $\mathbf{c} = \mathbf{Fx}$. Thus, an adversary hopes to output a vector $\mathbf{x}$, and a forged checkblock $\mathbf{c}'$ such that $h_G(\mathbf{c}') = h_G(\mathbf{Fx})$ and $\mathbf{c}' \neq \mathbf{Fx}$. Considering such an adversary, $\mathcal{A}'_G$, we define the probability that it succeeds in its forgery:

$$\mathsf{Adv}_1(\mathcal{A}'_G) = \Pr[g \leftarrow G;\ \mathbf{r} \leftarrow \mathbb{Z}_q^m;\ \mathbf{F} \leftarrow \mathcal{F};\ (\mathbf{x}, \mathbf{c}') \leftarrow \mathcal{A}'_G(g^{\mathbf{r}}, \mathbf{F}) : h_G(\mathbf{Fx}) = h_G(\mathbf{c}') \ \wedge\ \mathbf{Fx} \neq \mathbf{c}']$$

It is easy to see that given a PPT A'_G, we can construct a corresponding algorithm $\mathcal{A}_G$ that finds hash-collisions as in above. Given $g^{\mathbf{r}}$, $\mathcal{A}_G$ chooses a random $\mathbf{F}$ from the distribution $\mathcal{F}$ and then runs $(\mathbf{x}, \mathbf{c}') \leftarrow \mathcal{A}'_G(g^{\mathbf{r}}, \mathbf{F})$. Lastly, $\mathcal{A}_G$ outputs $(\mathbf{Fx}, \mathbf{c}')$. By construction of the two preceding random experiments, we can state quite simply:

$$\mathsf{Adv}_1(\mathcal{A}'_G) = \mathsf{Adv}_0(\mathcal{A}_G) \tag{5}$$

Note that we do not consider the properties of $\mathcal{F}$ here. Rather, the publisher's selection of $\mathbf{r}$ from the uniform distribution $\mathbb{Z}_q$ ensures the collision-resistance of the hash function.

Now consider our verifier function $\mathcal{V}_G$. $\mathcal{V}_G$ takes as input an $m \times t$ batch $\mathbf{C}$, the batch's supposed hash y, and a random t-vector $\mathbf{s}$ of l-bit numbers. It then should output 1 if $H_G(\mathbf{C}) = y$, and 0 otherwise. Define an adversary $\mathcal{A}''_G$ who attempts to trick $\mathcal{V}_G$ into wrongly verifying a batch of t checkblocks. As above, we define its likelihood of success in terms of randomized experiment:

$$\mathsf{Adv}_2(\mathcal{A}''_G) = \Pr[g \leftarrow G;\ \mathbf{r} \leftarrow \mathbb{Z}_q^m;\ \mathbf{F} \leftarrow \mathcal{F};\ (\mathbf{X}, \mathbf{C}') \leftarrow \mathcal{A}''_G(g^{\mathbf{r}}, \mathbf{F});\ \mathbf{s} \leftarrow \mathbb{Z}_{2^l}^t : \mathbf{FX} \neq \mathbf{C}' \ \wedge\ \mathcal{V}_G(\mathbf{C}', H_G(\mathbf{FX}), \mathbf{s}) = 1]$$

In this experiment, the adversary outputs his checkblock recipes as a batch of t n-vectors, or rather, as an $n \times t$ matrix $\mathbf{X}$, whose cells are in $\{0,1\}$. While the batch of checkblocks the verifier expects is $\mathbf{C} = \mathbf{FX}$, the adversary also outputs a forged batch of checkblocks $\mathbf{C}'$.

Consider cases in which $\mathcal{V}$ outputs 1. For $\mathcal{V}_G$ to verify $(\mathbf{C}', H_G(\mathbf{C}))$, it computes y and y' as given in Step 4 of the batch verification procedure. More succinctly:

$$y' = g^{\mathbf{rC's}},\ y = \prod_{j=1}^{t} h_G(\mathbf{c}_j)^{s_j}$$

If we were to run the same batch verification procedure on the original $\mathbf{C}$, we would find $y'' = g^{\mathbf{rCs}}$. We have that $y \equiv y' \bmod p$ because V output 1. We also have that $y \equiv y'' \bmod p$ because V will always verify $(\mathbf{C}, H(\mathbf{C}))$. Hence, $y' \equiv y'' \bmod p$. Because g is a generator of G, we claim that the exponents of y' and y'' are equivalent $\bmod q$. Rearranging:

$$\mathbf{r}(\mathbf{C} - \mathbf{C}')\mathbf{s} \equiv 0 \bmod q \tag{6}$$

That is, $\mathcal{V}_G$ will output 1 on input $(\mathbf{C}', H_G(\mathbf{C}))$ if and only if Equation 6 holds. Let $\mathbf{u} = \mathbf{r}(\mathbf{C} - \mathbf{C}')$. We consider two different cases: when $\mathbf{u} \equiv 0_{\mathbf{v}} \bmod q$ and when it is non-zero. In the first case, we rearrange to see that $\mathbf{rC} \equiv \mathbf{rC}' \bmod q$, therefore $g^{\mathbf{rC}} \equiv g^{\mathbf{rC}'}$, and by definition of H_G, $H_G(\mathbf{C}) = H_G(\mathbf{C}')$. If we assume such a batch-forging adversary $\mathcal{A}''_G$ to exist, then we can easily construct a derived PPT adversary $\mathcal{A}'_G$ that forges single checkblocks. $\mathcal{A}'_G$ would call $\mathcal{A}''_G$ as normal, and then find column j such that $\mathbf{Fx_j} \neq \mathbf{c_j}'$. Since $\mathbf{FX} \neq \mathbf{C}'$, such a column must exist. $\mathcal{A}'_G$ would then output $(\mathbf{x_j}, \mathbf{c_j}')$. This simple reduction shows that

$$\mathsf{Adv}_2(\mathcal{A}''_G) = \mathsf{Adv}_1(\mathcal{A}'_G) \text{ when } \mathbf{u} \equiv 0_{\mathbf{v}} \bmod q \tag{7}$$

We can now compose Equations 5 and 7. For any given $\mathcal{A}''_G$, we can construct a corresponding adversary $\mathcal{A}_G$ who finds collisions of the discrete-log CRHF in G such that $\mathsf{Adv}_2(\mathcal{A}''_G) = \mathsf{Adv}_0(\mathcal{A}_G)$. Recall that for all PPT $\mathcal{A}_G$, $\mathsf{Adv}_0(\mathcal{A}_G)$ is negligible in λ by the discrete log assumption. Hence for all PPT $\mathcal{A}''_G$:

$$\mathsf{Adv}_2(\mathcal{A}''_G) \leq \mathsf{negl}(\lambda) \text{ when } \mathbf{u} \equiv 0_{\mathbf{v}} \bmod q \tag{8}$$

In the second case, let $\mathbf{u} \not\equiv 0_{\mathbf{v}} \bmod q$. Again, by Equation 6, $\mathbf{us} \equiv 0 \bmod q$. If $\mathbf{s} = 0_{\mathbf{v}}$, then clearly $\mathbf{us} \equiv 0 \bmod q$, but such an $\mathbf{s}$ is selected only with probability 2^{-lt}. Otherwise, there exists at least one element of $\mathbf{s}$ that is non-zero. Call this element s_1 without loss of generality. We can write:

$$u_1 s_1 \equiv -\sum_{j=2}^{t} u_j s_j \bmod q$$

Since q is prime, u_1 has exactly one multiplicative inverse, and consequently there is exactly one such $s_1 \in \{0,1\}^l$ such that the above equation holds, and it is chosen with probability 2^{-l}. Thus:

$$\Pr\left[\mathbf{us} \equiv 0 \bmod q \;\middle|\; \mathbf{u} \not\equiv 0_{\mathbf{v}} \bmod q\right] = 2^{-l}(1 + 2^{-t}) \tag{9}$$

Combining the two independent cases in Equations 8 and 9 we can write the upper bound:

$$\mathsf{Adv}_2(\mathcal{A}''_G) \leq \mathsf{negl}(\lambda) + 2^{-l}(1 + 2^{-t}) \tag{10}$$

for all PPT $\mathcal{A}''_G$. Since $l \ll \lambda$, we conclude that the success of any PPT algorithm $\mathcal{A}''_G$ in making $\mathcal{V}_G$ incorrectly verify is negligible in the security parameter l.