Paradoxical Compression with Verifiable Delay Functions

Thomas Pornin

NCC Group, thomas.pornin@nccgroup.com

2 October, 2021

Abstract. Lossless compression algorithms such as DEFLATE strive to reliably process arbitrary inputs, while achieving compressed sizes as low as possible for commonly encountered data inputs. It is well-known that it is mathematically impossible for a compression algorithm to simultaneously achieve non-trivial compression on some inputs (i.e. compress these inputs into strictly shorter outputs) and to never expand any other input (i.e. guaranteeing that all inputs will be compressed into an output which is no longer than the input); this is a direct application of the "pigeonhole principle". Despite their mathematical impossibility, we show in this paper how to build such paradoxical compression and decompression algorithms, with the aid of some tools from cryptography, notably verifiable delay functions, and, of course, by slightly cheating.

1 Paradoxical Compression

The *pigeonhole principle* is the colloquial name for the general remark that, given two finite sets S_1 and S_2, there cannot exist an injective map from S_1 to S_2 if the cardinality of S_1 is strictly greater than that of S_2. The principle appears to have been used by mathematicians since at least the early 17th century[8] and was first formalized by Dirichlet two centuries later[5]. Dirichlet's metaphor involved drawers, which, through some later translation mishaps, led to the name "pigeonhole" and the description of the principle involving birds in a dovecote[10]. In rough terms, if you have more pigeons than holes, you cannot put each pigeon alone in a hole; there must be at least one hole where you will cram two pigeons together, or a pigeon with no hole to sit into.

This principle applies to *lossless compression algorithms*. We consider the set $\mathbb{B}$ of ordered finite sequences of bits; each sequence $x \in \mathbb{B}$ has a length denoted $\mathsf{len}(x)$, which is the number of bits in the sequence. For any integer $n \geq 0$, there are precisely 2^n possible bit sequences of length n, and $2^{n+1} - 1$ possible bit sequences of length at most n. A lossless compression algorithm is defined as a pair of computable functions C and D, each taking bit sequences as input and as output, with the following characteristics:

- C takes as input any bit sequence $x \in \mathbb{B}$, and outputs a corresponding bit sequence $C(x)$. As a computable function, C may be randomized, i.e. there is no requirement that for a given x, the exact same output $C(x)$ is always obtained.
- D takes as input a bit sequence $y \in \mathbb{B}$, and outputs a corresponding bit sequence $D(y)$; for some inputs y, D may instead return $\perp$, a symbolic value distinct from all bit sequences which represents decompression failure. For a given input y, D must always return the same output (i.e. the implementation of D may be randomized, but, as a function, it is deterministic).

– For any $x \in \mathbb{B}$, $D(C(x)) = x$. This is what "lossless" means: no information about x is lost in the compression process, and x can be recovered through decompression.
– For inputs x that are expected to occur in a given usage context, the average compressed length ($\mathsf{len}(C(x))$) is lower than the average uncompressed length ($\mathsf{len}(x)$).

Compression is useful under the assumption that "normal data" is not uniformly distributed; i.e., for a given bit length n, a small fraction of bit sequences of that length are much more likely to appear as inputs to C than all others, and these inputs may thus be encoded into a shorter format. Commonly used generic-purpose compression algorithms such as DEFLATE[4] (used in the well-known GZip and Zlib formats, and the PNG image format) exploit repeated sequences of input bits (or bytes), as well as non-uniform distribution of input code units, as is typical of text-based data formats, to achieve non-negligible compression ratios at moderate computational cost.

If compression can reduce the length of some inputs, it must necessarily increase the length of other inputs. We will call *paradoxical compression* a lossless compression algorithm (C, D) such that:

– For any $x \in \mathbb{B}$, $\mathsf{len}(C(x)) \leq \mathsf{len}(x)$.
– There exists at least one input $x_0 \in \mathbb{B}$ such that $\mathsf{len}(C(x_0)) < \mathsf{len}(x_0)$.

The second condition implies that C is not simply a length-preserving permutation (e.g. the identity): some data can really be "compressed". Notwithstanding, there needs not be many inputs that can be such compressed.

The pigeonhole principle expresses the fact that paradoxical compression is impossible (hence the name). Indeed, if $n = \mathsf{len}(C(x_0))$, then consider the set $\mathbb{B}_n$ of all bit sequences of length at most n: for any $x \in \mathbb{B}_n$, paradoxical compression ensures that $\mathsf{len}(C(x)) \leq \mathsf{len}(x) \leq n$, hence $C(x) \in \mathbb{B}_n$. The input x_0 is not part of $\mathbb{B}_n$ (by construction), and thus the cardinality of $S_1 = \mathbb{B}_n \cup \{x_0\}$ is strictly greater than the cardinality of $S_2 = \mathbb{B}_n$ (namely, $\#S_1 = 2^{n+1} = \#S_2 + 1$), and therefore there cannot be an injective map from S_1 to S_2. This implies that there must be two distinct inputs x and x' in S_1 that are compressed into the same output, i.e. $C(x) = C(x')$. The decompressor, applied on that shared output value, cannot return both x and x', which means that compression is not lossless.

Now that we have established that paradoxical compression is mathematically impossible, we will show in this paper how to achieve it.

Of course there is a cheat. We will subtly change the rules of the game. Consider the unfortunately not too rare situation of a scammer, peddler of some "infinite compression" scheme, selling an algorithm (usually as a software or hardware black box) that claims to be able to reduce the size of any input, and still decompress the result back into the original data with no loss. This is a claim much stronger than paradoxical compression (in which it is merely claimed that no input is made strictly longer, but some inputs might not be made strictly shorter). In an abstract model, the scammer is defeated by simply trying uniformly random inputs:

– Choose an input length n.
– Choose a uniformly random sequence x of n bits.
– Obtain $y = C(x)$ and then $z = C(y)$.
– Verify that $\mathsf{len}(z) < \mathsf{len}(y) < n$ and that $D(D(z)) = x$.

One of the tests in the last step will fail with probability at least 1/2; it suffices to run that experiment a few dozen times to reliably reveal the fraud[1]. In practice, scammers will deploy great creativity in trying to succeed at that test by getting out of the abstract model, e.g. by using an extra covert channel between the compressor and the decompressor (for instance, encoding some information in the output file name).

We can reuse this setup in order to (re)define paradoxical compression in a way that makes it possible. We convert it into a game between a defender (the scammer) and an attacker (the potential scam victim), in which the attacker tries to reveal the fraud.

The paradoxical compression game:

- The defender provides C and D as two systems. Each system can receive an input (an arbitrary bit sequence) and produces a corresponding output (a bit sequence); the decompressor D may also return $\perp$ for a given input x.
- The defender shows bit sequences x_0 and y_0 such that $y_0 = C(x_0)$, $x_0 = D(y_0)$, and $\mathsf{len}(y_0) < \mathsf{len}(x_0)$.
- No communication of any kind may occur between C and D except the attacker's requests described below.
- The attacker submits compression requests x_i to C, and decompression requests y_j to D, obtaining the corresponding output ($C(x_i)$ or $D(y_j)$, respectively). The attacker may submit many requests, and use the responses to previous requests in order to produce the next ones.
- The attacker wins the game if any of the following occurs:
  - $\mathsf{len}(C(x_i)) > \mathsf{len}(x_i)$ for some compression request x_i
  - $C(x_i) = y_j$ and $D(y_j) \neq x_i$ for some pair of requests (x_i, y_j)

We will say that we achieve paradoxical compression if we can design systems C and D that defeat the attacker in this game. It is worth noting the difference between this game and the pigeonhole principle: in the pigeon metaphor, we are now allowed to store several pigeons in the same hole, provided that the attacker does not catch us doing so. This makes the game winnable by the defender, if pigeonhole overcrowding is a rare case and the defender can reasonably expect that the attacker will not be able to hit one of these problematic situations.

This metaphor explains why, in the infinite compression scam, the attacker always wins: if the defender promises that $\mathsf{len}(C(x_i)) < \mathsf{len}(x_i)$ (instead of merely $\mathsf{len}(C(x_i)) \leq \mathsf{len}(x_i)$ in the case of paradoxical compression), then there are too many tormented volatiles for their plight to remain unnoticed. As explained above, the attacker can choose a random x_i, then use compression requests x_i and $x_{i+1} = C(x_i)$, then decompression requests $y_j = C(x_{i+1})$ and $y_{j+1} = D(y_j)$; each such group of four requests leads to a win with probability at least 1/2.

On the other hand, if we only claim paradoxical compression, then the probability of attacker's success can be made much lower. This, of course, relies on implicit assumptions on

[1] The use of two nested compression calls is needed to avoid making hypotheses on the selection of the length n: if the scammer can guess in advance the length n we are going to choose, then they could make a compression function that injectively encodes $2^n - 1$ of the 2^n possible inputs of length n into the $2^n - 1$ sequences of length less than n, lowering the risk of fraud detection to 2^{-n}. With two compression calls, we ensure that z should be at least *two* bits shorter than x, as per the scammer's claims, which greatly increases the minimal probability of detection.

the number of requests: if the attacker can make an arbitrarily high number of requests, then it suffices to request all possible bit sequences x_i (in order of increasing length, and lexicographic order for a given length) until $C(x_i)$ matches a previously obtained $C(x_j)$ for some $x_j \neq x_i$. At that point, decompressing the common $C(x_i) = C(x_j)$ value might yield x_i or x_j, but not both, and the attacker wins.

2 Using a Shared Secret Key

We present here a concrete instantiation of paradoxical compression, built under an additional assumption: the compressor C and decompressor D share a secret key K that the attacker does not know. This is a restrictive hypothesis but it can match some usage contexts, e.g. mass storage compression scenarios (the attacker is a "normal user" who can choose arbitrary file contents to store, but the compressor and decompressor are privileged tasks in the operating system).

Paradoxical compression with a shared secret key:

Setup:

- (C_0, D_0) is a normal lossless compression scheme such as DEFLATE[4].
- K is a shared secret between the compressor and decompressor.
- MAC is a stateless deterministic message authentication code, e.g. HMAC[6], that uses K as key; its output has a known fixed length MAClen.
- enc is a function that encodes an unsigned integer value over clen bits with a fixed convention (e.g. big-endian); clen is chosen to be "somewhat large" (at least 64 bits in practice).

Compressor: On input $x \in \mathbb{B}$:

1. If $\mathsf{len}(C_0(x)) \leq \mathsf{len}(x) - \mathsf{clen} - \mathsf{MAClen}$:
   - Return $C(x) = C_0(x) \parallel \mathsf{enc}(0) \parallel \mathsf{MAC}(C_0(x) \parallel \mathsf{enc}(0))$.
2. Otherwise, if $x = d \parallel \mathsf{enc}(c) \parallel \mathsf{MAC}(d \parallel \mathsf{enc}(c))$ for some $d \in \mathbb{B}$ and integer $c \geq 0$:
   - Return $C(x) = d \parallel \mathsf{enc}(c+1) \parallel \mathsf{MAC}(d \parallel \mathsf{enc}(c+1))$.
3. Otherwise, return $C(x) = x$.

Decompressor: On input $y \in \mathbb{B}$:

1. If $y = d \parallel \mathsf{enc}(0) \parallel \mathsf{MAC}(d \parallel \mathsf{enc}(0))$ for some $d \in \mathbb{B}$:
   - Return $D(y) = D_0(d)$.
2. Otherwise, if $y = d \parallel \mathsf{enc}(c) \parallel \mathsf{MAC}(d \parallel \mathsf{enc}(c))$ for some $d \in \mathbb{B}$ and integer $c > 0$:
   - Return $D(y) = d \parallel \mathsf{enc}(c-1) \parallel \mathsf{MAC}(d \parallel \mathsf{enc}(c-1))$.
3. Otherwise, return $D(y) = y$.

The compressor obviously fulfills the rule of length non-extension: the returned value $C(x)$ is never longer than the input x. We now have to explain why the decompressor D correctly returns the original x given $C(x)$:

– If $C(x) = d \parallel \mathsf{enc}(0) \parallel \mathsf{MAC}(d \parallel \mathsf{enc}(0))$, as matched by decompressor case 1, then $C(x)$ cannot be an output of the compressor case 3, since that would mean that $x = d \parallel \mathsf{enc}(0) \parallel \mathsf{MAC}(d \parallel \mathsf{enc}(0))$, which would have been matched by the compressor case 2. Similarly, it cannot be an output of the compressor case 2, since that case produces only counter values $c + 1 > 0$ (this assertion is only computationally true, see below). Thus, it must be the output of compressor case 1, and decompressor case 1 properly returns $D_0(d) = x$.
– If $C(x) = d \parallel \mathsf{enc}(c) \parallel \mathsf{MAC}(d \parallel \mathsf{enc}(c))$ for some $c > 0$, as matched by decompressor case 2, then it must similarly be an output of compressor case 2, and the decompressor correctly returns the original x (thanks to the fact that the MAC output is deterministic).
– Otherwise, the provided $C(x)$ can be only an output of compressor case 3, i.e. $C(x) = x$, and this is the value that decompressor case 3 correctly returns.

The analysis above relies on the idea that compressor case 2, where the counter value $c+1$ is encoded by enc, *must* be matched by the decompressor case 2 but not case 1, i.e. if $c \geq 0$, then $\mathsf{enc}(c + 1)$ *must not* be equal to $\mathsf{enc}(0)$. This is of course true for all values of c from 0 to $2^{\mathsf{clen}} - 2$, but not for $c = 2^{\mathsf{clen}} - 1$. In the latter case, the counter "wraps around" and the encoding of $c + 1$ over clen bits either fails, or is truncated to the value zero; in either case, the attacker wins the game. Thus, the construction hinges on the impossibility for the attacker to produce a compressor input $x = d \parallel \mathsf{enc}(2^{\mathsf{clen}} - 1) \parallel \mathsf{MAC}(d \parallel \mathsf{enc}(2^{\mathsf{clen}} - 1))$. This impossibility comes from the following:

– The MAC construction is supposed to be secure, i.e. not to allow forgeries. The only way to obtain a valid $\mathsf{MAC}(t)$ value for any t is to use the key K, which is known only to the compressor and the decompressor.
– The MAC outputs computed by the decompressor are always over data containing the counter value $c - 1$, *after* having verified that a value containing a MAC over counter value c was provided. Thus, the decompressor cannot be used to obtain a MAC over a counter value greater than what was already known.
– The compressor will produce a MAC output over counter value $c+1$, but only after having verified that the provided input contained a counter value c with a valid MAC, which must itself have been obtained beforehand. Therefore, in order to obtain $\mathsf{MAC}(d \parallel \mathsf{enc}(2^{\mathsf{clen}} - 1))$, at least 2^{clen} requests must have been made to the compressor C; these requests must moreover be made sequentially, since each is over the output of the previous request. Since clen was chosen to be "somewhat large", the compressor cannot practically process all these requests in a feasible amount of time.

In the pigeon metaphor, the two "pigeons" $x_1 = x$ and $x_2 = C_0(x) \parallel \mathsf{enc}(2^{\mathsf{clen}} - 1) \parallel \mathsf{MAC}(C_0(x) \parallel \mathsf{enc}(2^{\mathsf{clen}} - 1))$ are stored (i.e. compressed) into the same hole $y = C_0(x) \parallel \mathsf{enc}(0) \parallel \mathsf{MAC}(C_0(x) \parallel \mathsf{enc}(0))$, and the decompressor returns x_1 on input y, not x_2. However, the attacker cannot produce the problematic pigeon x_2, since that would require either a MAC forgery, or coercing the compressor into producing x_2 as an output, the latter case being unreachable because of the finite computing power of the compressor. We furthermore note that the computing power of the attacker does not matter here: the impossibility to reach $2^{\mathsf{clen}} - 1$ comes from the compressor's limitations, not the attacker's. Moreover, the inherent sequentiality of the requests that lead to high counter values c implies that no attack speed-up ensues if there are multiple instances of the compressor C with the same secret key K.

In practice, one may achieve proper security with a 128-bit MAC output, and a 64-bit counter. This implies an overhead of only 192 bits (24 bytes) over the "raw" compression scheme (C_0, D_0):

- If $\mathsf{len}(C_0(x)) > \mathsf{len}(x) - 192$, then C returns x and gains nothing on the length (compression does not yield enough room for our custom header with the counter and the MAC).
- Otherwise, $\mathsf{len}(C(x)) = \mathsf{len}(C_0(x)) + 192$.

While the usage context, with a shared secret key between compressor and decompressor, is restrictive, this construction is practical and has low computational overhead, MAC computations being normally quite fast.

3 Using a Verifiable Delay Function

We can now remove the shared key requirement, by replacing the MAC with a verifiable delay function (VDF). The concept of VDFs was described in 2018[1] with some proposals for the concrete instantiation; additional constructions with improved characteristics (shorter output, lower proof computation overhead) were shortly after proposed by Pietrzak[9] and Wesolowski[11] (see also [2] for a comparison of these two constructions). In our concrete practical instantiation of keyless paradoxical compression, we will use Wesolowski's VDF.

A VDF consists of three algorithms:

- $\mathsf{Setup}(\lambda) \rightarrow p$ produces (randomly) some public parameters p used by the other algorithms; λ is a security parameter.
- $\mathsf{Eval}(p, T, e) \rightarrow (f, \pi)$ works over an input message $e \in \mathbb{B}$ and a time factor T (an integer), and outputs a value f along with a proof π.
- $\mathsf{Verify}(p, T, e, f, \pi) \rightarrow \{\text{true, false}\}$ checks that (f, π) is a correct evaluation of the VDF over e (i.e. (f, π) is a possible output of $\mathsf{Eval}(p, T, e)$).

We slightly diverge from the formal definition in [1] in that we suppose that the security parameters can accept several time factor values, provided dynamically to Eval and Verify; in the terminology of [1], we expect the VDF to be *incremental*. The definition of a VDF requires *uniqueness*, i.e. that for given input values p, T and e, a single output f may (practically) be found such that $\mathsf{Verify}(p, T, e, f, \pi) = \text{true}$; in our case, we will also require π to be similarly unique for given p, T and e. We will call this last property *proof uniqueness*. Finally, we also need VDF outputs (f, π) to be encodable into a bit sequence with a fixed length (possibly with some ad hoc padding).

The point of a VDF is that computing Eval requires $O(T)$ sequential operations, thus taking time $O(T)$, and that time cannot be substantially reduced by using more compute units (i.e. the computation cannot be performed in parallel); on the other hand, Verify should be fast. The VDF output thus represents a proof that some quantifiable physical time must have been spent for its computation over the input data e. The ability to fine-tune the expectation on the minimal delay for evaluation with a given work factor T is an important and active research area, targeting use in applications such as blockchain consensus protocols. However, in our case, we will use a VDF in a somewhat different context that entails different requirements.

In the paradoxical compression method described in section 2, the success of the construction relies on compressed outputs including a counter value c that keeps track of how many times the output has been re-compressed; high counter values are made unattainable by virtue of being authenticated with a MAC, and the only entities able to produce new MAC values structurally refuse to try new counter values except in small increments. Since high values cannot be practically reached by attackers, a fixed-size counter can be used and the counter overflow issue can be ignored. In a keyless solution, we replace the MAC with a VDF; instead of a counter increment policy enforced by the MAC key holders, we will use the counter c as work factor, and rely on the VDF sequentiality to make counter overflow conditions unreachable.

This leads to the following keyless paradoxical compression:

Keyless paradoxical compression with a VDF:

Setup:

- (C_0, D_0) is a normal lossless compression scheme such as DEFLATE[4].
- $(\mathsf{Setup}, \mathsf{Eval}, \mathsf{Verify})$ is an incremental VDF which fulfills proof uniqueness. The VDF public parameters (output of Setup) are p; they are generated once and hard-coded into both the compressor and the decompressor. The output of Eval can be encoded over exactly VDFlen bits.
- enc is a function that encodes an unsigned integer value over clen bits with a fixed convention (e.g. big-endian); clen is chosen to be large enough that $\mathsf{Eval}(p, 2^{\mathsf{clen}}-1, e)$ is computationally too expensive to produce for any input $e \in \mathbb{B}$.

Compressor: On input $x \in \mathbb{B}$:

1. If $\mathsf{len}(C_0(x)) \leq \mathsf{len}(x) - \mathsf{clen} - \mathsf{VDFlen}$:
   - Return $C(x) = C_0(x) \parallel \mathsf{enc}(0) \parallel \mathsf{Eval}(p, 0, C_0(x))$.
2. Otherwise, if $x = d \parallel \mathsf{enc}(c) \parallel (f, \pi)$ for some $d \in \mathbb{B}$, integer $c \geq 0$, and (f, π) a syntactically correct VDF output such that $\mathsf{Verify}(p, c, d, f, \pi) = \text{true}$:
   - Return $C(x) = d \parallel \mathsf{enc}(c+1) \parallel \mathsf{Eval}(p, c+1, d)$.
3. Otherwise, return $C(x) = x$.

Decompressor: On input $y \in \mathbb{B}$:

1. If $y = d \parallel \mathsf{enc}(0) \parallel (f, \pi)$ for some $d \in \mathbb{B}$ and (f, π) a syntactically correct VDF output such that $\mathsf{Verify}(p, 0, d, f, \pi) = \text{true}$:
   - Return $D(y) = D_0(d)$.
2. Otherwise, if $y = d \parallel \mathsf{enc}(c) \parallel (f, \pi)$ for some $d \in \mathbb{B}$, integer $c > 0$, and (f, π) a syntactically correct VDF output such that $\mathsf{Verify}(p, c, d, f, \pi) = \text{true}$:
   - Return $D(y) = d \parallel \mathsf{enc}(c-1) \parallel \mathsf{Eval}(p, c-1, d)$.
3. Otherwise, return $D(y) = y$.

This construction achieves paradoxical compression for reasons similar to the keyed construction from section 2:

– Decompression cases 1, 2 and 3 undo the operations of compression cases 1, 2 and 3, respectively. In particular, decompression case 2 rebuilds the exact same bit sequence that was previously produced by compression case 1 or 2 thanks to the uniqueness and proof uniqueness of the VDF (since only a single output f and single proof π can be practically produced for input d and work factor c, the same f and π must be generated as the ones previously produced by the compressor).
– The attacker is defeated insofar as the counter maximal value $2^{\mathsf{clen}} - 1$ cannot be reached. This is ensured by making the counter size clen large enough; the size depends on the VDF internals, but in our practical instantiation (described below) 128 bits ought to be enough. We note here that since the security relies on the sequentiality of the VDF, not by a policy enforced by a defender's system, the attacker may use the most powerful sequential computing unit that can be found, possibly quite faster than the system on which the compressor will run. Therefore, the counter length clen should be defined with some extra margin.

3.1 Instantiation with Wesolowski's VDF

In Wesolowski's VDF[11], the public parameters p contain a big composite integer N whose factorization is unknown; in concrete terms, N is the modulus of an RSA key. Setup discards the factors of N; it is assumed that nobody knows them. This is a case of a *trusted setup*, a usually undesirable property since it raises the suspicion that the generating system may have kept a copy of the private factors; this drawback can be mitigated with procedural means (an audited key ceremony) or multi-party computation protocols[3]. In the rest of this section, N is a 2048-bit RSA modulus, generated with a normal RSA key pair generator, the private factors being discarded.

Wesolowski's VDF uses the group $\mathbb{G}^+$ of (unordered) pairs $\{u, -u\}$ for u a non-zero integer modulo N; the group operation combines $\{u, -u\}$ and $\{v, -v\}$ into $\{uv, -uv\}$. A group element $\{u, -u\}$ is uniquely represented by its component which is, as an integer, in the 1 to $(N-1)/2$ range (inclusive). For all practical purposes, this is the group of invertible integers modulo N, with the multiplication as group law, but we do not care about the difference between u and $-u$, and values are normalized into the "lower half" of the range of integers modulo N. This use of $\mathbb{G}^+$ fulfills the *low order assumption* that ensures the security of the construction[2].

H is a hash function with output in $\mathbb{G}^+$. A simple choice for H is to use a XOF such as SHAKE[7] to produce a value with the size of N (256 bytes for a 2048-bit modulus), which is then interpreted as a big integer with a given convention (e.g. unsigned big-endian encoding) and reduced modulo N, then normalized as per the rules of $\mathbb{G}^+$. We ignore the conceptually possible cases of the output of SHAKE being non-invertible modulo N (finding such a case would constitute a preimage attack on SHAKE, a factorization attack on N, or both).

Another hash function h is used to sample a random prime of size up to 256 bits. A simple instantiation is to first obtain a 256-bit output from a classic hash function such as SHA3-256 (or SHAKE again, with a 256-bit output), then interpret that 256-bit string as an integer t, and find the smallest prime $\ell \geq t$.

We assume that an unambiguous unique fixed-length encoding is defined for all values. For instance, N can be encoded over 2048 bits in unsigned big-endian conventions; any el-

ement of $\mathbb{G}^+$ is normalized into an integer in the 1 to $(N-1)/2$ range and can be encoded over 2047 bits, again in unsigned big-endian convention.

For public parameters p (i.e. the modulus N), input message e and time factor T, the VDF output $\mathsf{Eval}(p, T, e) = (f, \pi)$ is computed as follows:

1. Compute $g = H(e \parallel N)$.
2. Compute $f = g^{2^T}$. This can be done with T successive squarings modulo N; it is postulated that no faster method exists without knowing the prime factors of N.
3. Compute $\ell = h(g \parallel N \parallel T \parallel f)$.
4. Compute the proof $\pi = g^q$ for the integer $q = \lfloor 2^T / \ell \rfloor$.

Though q can be too large to comfortably fit in memory, the computation of π in a classic square-and-multiply algorithm only needs the bits of q in high-to-low order, and they can be obtained by running the schoolbook long division algorithm of 2^T by ℓ (see [2], section 3 for some details).

The verification process $\mathsf{Verify}(p, T, e, f, \pi)$ works as follows:

1. Compute $g = H(e \parallel N)$.
2. Compute $\ell = h(g \parallel N \parallel T \parallel f)$.
3. Compute $r = 2^T \bmod \ell$ (note: this implies that $2^T = q\ell + r$).
4. Compute $f' = \pi^\ell g^r$ in $\mathbb{G}^+$.
5. Return true if $f = f'$, false otherwise.

It can be proven that this process fulfills the expected VDF properties (see [11,2]). Proof uniqueness is also achieved because finding two different π and π' such that $\pi^\ell = \pi'^\ell$ would imply that π/π' has order ℓ in $\mathbb{G}^+$, making ℓ a prime factor of the unknown order of $\mathbb{G}^+$. Finding such a ℓ, computing an ℓ-th root of unity modulo N, and finding a preimage of ℓ by the hash function h are all considered infeasible if N was generated properly and h was built over a secure hash function.

Using a 2048-bit modulus and a 128-bit work factor (i.e. compression counter c), the total size overhead of this construction is 128 + 2047 + 2047 = 4222 bits (slightly over half a kilobyte). Whether such an overhead is acceptable depends on the usage context.

We implemented this algorithm in C#; source code can be found on:

https://github.com/pornin/paradox-compress/

3.2 Denial-of-Service attacks and updatable VDFs

The main drawback of the instantiation with Wesolowski's VDF is that it can raise the computational cost of compression and decompression. If the attacker spends once the cost of computing Eval over a given input e with a relatively high work factor T, yielding a valid VDF output (f, π), then every time the input $e \parallel \mathsf{enc}(T) \parallel (f, \pi)$ is submitted to the compressor (or the decompressor), the compressor will need to perform the computation of the VDF with work factor $T+1$ (or $T-1$ for the decompressor), with cost $O(T)$. This can lead to denial-of-service (DoS) attacks in contexts where the attacker can submit chosen inputs to a compressor or decompressor system that is not nominally in reach of the attacker.

To mitigate such attacks, a nice solution would be an *updatable VDF*, i.e. a VDF such that computing $\mathsf{Eval}(p, T+1, e)$ or $\mathsf{Eval}(p, T-1, e)$ would be inexpensive, given the output

of Eval(p, T, e). Unfortunately, Wesolowski's VDF is not updatable in that sense. Updatability does not seem to inherently contradict the definition of a VDF, but, since it was not envisioned (yet) as a potentially useful property, no effort has been made to find VDFs that offer that feature. We can note that given g^{2^T}, computing $g^{2^{T+1}}$ is already inexpensive (it is a matter of a single modular squaring); however, the same cannot be said about computing the proof for time factor $T + 1$, since it will use a different and unpredictable value for the prime ℓ. As for updates in the other direction (for decompression), there is no known method for computing $g^{2^{T-1}}$ from g^{2^T} that is faster than starting again from g, with cost $O(T)$ and disregarding the provided g^{2^T}.

4 Conclusion

We presented methods to instantiate a nominally impossible algorithm: a lossless compression algorithm that can reduce the size of some inputs, but never increases the size of any input. In practical terms, this is useless.

Indeed, compression is useful only if the overall context can take advantage of a smaller but dynamically obtained data size; such contexts normally involve encoding formats with some metadata such as a "length" field. If the maximum allowed data length (m) is lower than the greatest integer that can be stored in that "length" slot (usually $2^k - 1$ for a k-bit field), then it is possible to implement compression without exceeding overall message size limits by using a special length value of $m + 1$ in some cases:

- If the data has length m and cannot be compressed by at least 1 bit, then the uncompressed data can be used as the payload, and the length field set to the special value $m + 1$.
- Otherwise, if the data has length at most $m - 1$ and cannot be compressed by at least 1 bit, then an extra bit of value 0 can be added as a header, leading to an at most m-bit payload.
- Otherwise, an extra bit of value 1 can be added as a header to the compressed data (of length as most $m - 2$), yielding a payload of length at most $m - 1$.

In this example, compression is achieved without increasing the maximum payload length. The pigeonhole principle is not contradicted here: the real payload is sometimes enlarged, but we can smuggle the extra information in the header, as the special length value $m + 1$.

This simple example shows that there usually are encoding tricks that allow for compression to be fitted inside the limitations of an existing protocol. For paradoxical compression to be useful, it takes a specific context in which such information smuggling in metadata is not doable for some reason (e.g. interoperability with legacy systems).

Despite the apparent futility of the exercise, the paradoxical compression constructions described here nevertheless have some virtues:

- They show how a slight model shift can turn an impossible functionality into an implementable one. Going from "this cannot mathematically be done" to "you cannot catch me doing something wrong" opens potentially unbounded possibilities. Arguably, most of the field of cryptography fits in the gap between these two notions.
- At the same time, the same model shift highlights the hopelessness of "infinite compression" schemes, which are not only impossible conceptually, but also necessarily detectable with the most simple and ham-fisted attacker challenges (random messages!).

– Paradoxical compression is yet another field of application of VDFs, which shows how these primitives can be versatile, useful and worth extra research efforts.
– The VDF property of updatability has been defined, and finding a VDF construction that provides it, or proving that it cannot be done, is a new open research question.

Acknowledgements

We thank Parnian Alimi, Marie-Sarah Lacharité and Eric Schorn for useful discussions and reviewing of draft versions of this paper.

No corporeal bird was harmed in the making of this article.

References

1. D. Boneh, J. Bonneau, B. Bünz and B. Fish, *Verifiable Delay Functions*, Advances in Cryptology - CRYPTO 2018, Lecture Notes in Computer Science, vol. 10991, pp. 757-788, 2018.
2. D. Boneh, B. Bünz and B. Fisch, *A Survey of Two Verifiable Delay Functions*, https://eprint.iacr.org/2018/712
3. M. Chen, R. Cohen, J. Doerner, Y. Kondi, E. Lee, S. Rosefield and a. shelat, *Muliparty generation of an RSA modulus*, Advances in Cryptology – CRYPTO 2020, Lecture Notes in Computer Science, vol. 12172, pp. 64-93, 2020.
4. P. Deutsch, *DEFLATE Compressed Data Format Specification version 1.3*, https://datatracker.ietf.org/doc/html/rfc1951
5. P. G. L. Dirichlet, *Recherches sur les formes quadratiques à coefficients et à indéterminées complexes*, Journal für die reine und angewandte Mathematik, vol. 24, pp. 291-371, 1842.
6. H. Krawczyk, M. Bellare and R. Canetti, *HMAC: Keyed-Hashing for Message Authentication*, https://datatracker.ietf.org/doc/html/rfc2104
7. Information Technology Laboratory, *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, National Institute of Standard and Technology, FIPS 202, 2015.
8. J. Leucheron, *Selectæ Propositiones in Tota Sparsim Mathematica Pulcherrimæ*, Gasparem Bernardum, 1622.
9. K. Pietrzak, *Simple Verifiable Delay Functions*, Innovations in Theoretical Computer Science Conference (ITCS), pp. 60:1-60:15, 2019.
10. B. Rittaud and A. Heeffer, *The pigeonhole principle, two centuries before Dirichlet*, Mathematical Intelligencer, vol. 36, issue 2, pp. 27-29, 2014.
11. B. Wesolowski, *Efficient Verifiable Delay Functions*, Advances in Cryptology - EUROCRYPT 2019, Lecture Notes in Computer Science, vol. 11478, pp. 379-407, 2019.