Changing Base without Losing Space

Yevgeniy Dodis
New York University

Mihai Pătraşcu
AT&T Labs

Mikkel Thorup
AT&T Labs

April 11, 2010

Abstract

We describe a simple, but powerful local encoding technique, implying two surprising results:

1. We show how to represent a vector of n values from Σ using $\lceil n \log_2 \Sigma \rceil$ bits, such that reading or writing any entry takes $O(1)$ time. This demonstrates, for instance, an "equivalence" between decimal and binary computers, and has been a central toy problem in the field of succinct data structures. Previous solutions required space of $n \log_2 \Sigma + n / \log^{O(1)} n$ bits for constant access.

2. Given a stream of n bits arriving online (for any n, not known in advance), we can output a *prefix-free* encoding that uses $n + \log_2 n + O(\log \log n)$ bits. The encoding and decoding algorithms only require $O(\log n)$ bits of memory, and run in constant time per word. This result is interesting in cryptographic applications, as prefix-free codes are the simplest counter-measure to extensions attacks on hash functions, message authentication codes and pseudorandom functions. Our result refutes a conjecture of [Maurer, Sjödin 2005] on the hardness of online prefix-free encodings.

1 Introduction

Suppose we want to represent a vector $A[1\,..\,n]$, where each element comes from some finite alphabet Σ. The optimal space is $\lceil n \log_2 \Sigma \rceil$ bits,[1] and it can be achieved by treating the vector as a big number between 0 and $\Sigma^n - 1$, and converting this number to binary. Unfortunately, this encoding has very poor locality: (1) we cannot encode a stream of symbols with a low-memory algorithm; (2) we cannot read or write a single element $A[i]$ faster than decoding the entire vector.

In this paper, we give a surprisingly simple technique that solves both of these problems optimally. We can represent A using $\lceil n \log_2 \Sigma \rceil$ bits, while reading/writing $A[i]$ in constant time. Additionally, given a stream of symbols in Σ, we can encode (and later decode) it into a binary stream of optimal size, while using $O(\log(n\Sigma))$ bits of memory and constant processing time per symbol. As an unexpected application of this result, we show how to encode/decode a stream of a-priori unknown length in a prefix-free manner, using essentially optimal overhead and memory.

1.1 Representing a Vector

The field of succinct data structures is preoccupied with representing data in close to optimal space, while still supporting useful queries. Representing a vector to support reads and writes to individual entries is perhaps the most foundational problem in the field, and thus has been a toy problem of particular interest.

The naïve way of supporting local access is to encode each element separately using $\lceil \log_2 \Sigma \rceil$ bits (cf. Hamming codes versus arithmetic codes). This has a redundancy of $\Omega(n)$ bits when Σ is not a power of two; for instance, it wastes around $0.68n$ bits for decimal digits, $\Sigma = \{0, 1, \dots, 9\}$.

Standard techniques in succinct data structure, dating back to [Jac89], could achieve constant access time using $n \log_2 \Sigma + O(n/\log n)$ bits of space. More recently, Golynski et al. [GGG+07] proposed an encoding with $O(n/\log^2 n)$ redundancy and constant access time. Pătraşcu [Păt08] showed how to use recursion in succinct encodings, and achieved query time $O(t)$ with a redundancy of $O(n/(\frac{\log n}{t})^t)$. Setting $t = O(1)$, we get redundancy $n/\log^{O(1)} n$ with constant query time.

For a growing body of problems, we have strong redundancy lower bounds of $\Omega(n/t)$ [GM03], $\Omega(n/t^2)$ [Gol09], or $n/\log^{O(t)} n$ [PV10]. Combining this with the non-locality of known optimal encodings (e.g., arithmetic coding described in Section 1.3), one may conjecture that any representation of a vector will require some nontrivial trade-off between the redundancy and query time.

In this paper, we show that this is not the case, and a surprising optimal result is possible:

Theorem 1. *On a Word RAM, one can represent a vector $A[1\,..\,n]$ of elements from a finite alphabet Σ using $\lceil n \log_2 \Sigma \rceil$ bits, such that any element of the vector can be read or written in constant time. The data structure requires $O(\log n)$ precomputed word constants.*

From a conceptual perspective, the statement that a binary computer can represent data in any radix with no penalty has an inherent appeal.

From a theoretical perspective, our result forms a nice contrast with the lower bounds of [PV10]. An overwhelming majority of succinct data structures rely crucially on the so-called rank/select problem. But it is proved in [PV10] that the redundancy needs to be at least $n/\log^{O(t)} n$, essentially matching the upper bound [Păt08]. Two scenarios are plausible: either (1) the use of rank/select is inherent, and we need stronger lower bound techniques that apply to a broader class of problems; or (2) one can surpass the rank/select barrier by alternative techniques.

In this paper, we illustrate scenario (2) for a natural, central question. One can hope that this opens the door to further improvements. For example, an important target is the dictionary problem

[1] When clear from context, we will abuse the notation and use Σ to denote both the alphabet set and its cardinality.

(a.k.a. set membership) — perhaps the most important remaining open problem in this field — where the state-of-the-art remains [Păt08].

Further discussion. The model of computation throughout this paper is the Word RAM, which is meant to formalize what is possible in imperative programming languages. The memory allows random access, and is organized into words of w bits, where $w = \Omega(\log n)$ in order to allow indices and pointers. A standard set of unit-cost operations is specified (in this paper, we only require addition, multiplication, and division).

In the (unrealistic but mathematically cleaner) bit-probe model, Viola [Vio09] has recently shown that, if the query is allowed t bit probes, the redundancy must be at least $n/2^{O(t)}$. Our new result provides a matching upper bound: simply simulate the actions of a RAM with t-bit words.

It is unclear whether the abstract vector-representation problem has direct applications in practice. Applications to compressed Bloom filters are suggested by [Mit02]. In addition, extreme compression is of interest in current database applications. In many databases, columns have few common possibilities (as few as 3-5), so they can be compressed to an alphabet with these possibilities plus a letter indicating "other." After pruning many rows using indexes, database queries degenerate into linear scans over ranges, and such scans are often memory-bound. Thus, it is worth using a (slightly) more CPU intensive algorithm for tighter compression of the records. In this context, not wasting one bit for small fields might be valuable. We mention that algorithms that exploit packing small fields into bits *are* currently deployed in a large commercial database [RS06, HRSD07]; in fact, this was the initial motivation behind our theoretical investigation.

1.2 Online Prefix-Free Encoding

Suppose one is to represent some n bits of information using a prefix-free code, where n is not fixed (i.e. it should be implicitly included in the code). In information theory, this task is known as *universal coding*, whereas in Computer Science, the term "prefix-free" is more often used (ambiguously). The term "self-delimiting code" is also in use.

The textbook solution to this problem is given by Elias codes. For example, to encode a vector of n bits using the Elias delta-code, one can write: (1) as many zeros as there are bits in $\log_2 n$, followed by a one; (2) the number n written in binary, using as many bits as indicated in part 1; (3) the n bits of data. This achieves $n + O(\log n)$ bits, for any n. Applying this idea recursively, one obtains the Elias omega-code, which represents n input bits using $n + \log_2 n + \log_2 \log_2 n + \cdots + O(\log^\star n)$ output bits. This bound is optimal.

Prefix-free encodings are essential for communicating or storing any type of data, except fixed-size records. Less obviously, prefix-free encodings are often essential to ensure security of iterative constructions in cryptography, such as CBC-MAC [BRK94] and cascade constructions [GGM86, BCK96a, CDMP05, MS05]. In particular, prefix-freeness gives the simplest provably-secure counter-measure to various forms of *extension attacks* on hash functions, message authentication codes and pseudorandom functions, as discussed in Appendix A.

In many natural applications, including those in cryptography, the message comes online, as a stream of bits, and the length of the stream is not known in advance. Elias codes are unusable in this setting. Instead, one wishes to design a simple code that can be encoded and decoded in one pass by algorithms maintaining small state (ideally, $O(\log n)$ bits), and using lower overhead (ideally, $O(1)$ time per input word).

In practice, one typically assumes that the input comes in large blocks, say, $b = 128$ bits; the last block may be padded if it is incomplete. Then, one can append one bit after each block, indicating whether we have reached the end of file. With $b = 128$, this code has an overhead of almost 1%, which is acceptable, but a fairly steep price to pay just for encoding the length of the stream. More problematically, many cryptographic standards include workarounds that eliminate the need for prefix-

freeness, creating theoretical and practical burdens.

In theory, the naïve block-based solution gives a linear trade-off between redundancy and the space of the algorithm, by buffering b bits (e.g., one block) of input, and including an end-of-file marker before each block.

Maurer and Sjödin [MS05] conjectured that a linear redundancy might be necessary for small-space online prefix-free encoding. This would explain why cryptographic standards tend to use ad hoc solutions instead, since a proportional increase in the stream size is unacceptable. Fortunately, we disprove this conjecture, by constructing a simple prefix-free code with ideal space and running time.

As the first step, we point out the fallacy in the linear trade-off between space and redundancy: thinking in terms of integral bits. Let us imagine that the stream consists of n blocks of b bits each, and we are guaranteed that $n \leq 2^b$. We can augment the alphabet of $[2^b]$ with an end-of-file symbol, EOF, making the stream prefix-free. If we can *efficiently* move from this alphabet of 2^b+1 to a binary alphabet (not wasting one bit per block), then the stream will take $(n+1)\log_2(2^b+1)$ bits. This means an overhead of $(n+1)\log_2(2^b+1) - n \cdot b = n\log_2(1+\frac{1}{2^b}) + b + O(\frac{1}{2^b}) = O(\frac{n}{2^b}) + b$. Since we were guaranteed $n \leq 2^b$, the overhead is $b + O(1)$. To remove this assumption, we can use slowly growing blocks ("guessing" n up to a factor of two). This gives an overhead of $O(\log n)$.

In this paper, we give an efficient online algorithm that converts base 2^b+1 to base 2^b (or, equivalently, to binary), thus refuting the conjecture of [MS05]. In fact, we describe two solutions in Sections 2 and 5.

The first algorithm assumes $b \geq 2\log_2 n + 1$ and wastes at most three blocks. It benefits from extreme simplicity and very fast running time, so we expect it to be the method of choice in practical implementations. A slightly more complicated version of the algorithm achieves optimal encoding size ($n+1$ blocks) as long as $b \geq 2\log_2 n + 2$.

Our algorithm has excellent locality properties. During normal operation, the algorithm outputs a pair of blocks for each pair of blocks it reads. This takes constant processing time, and the memory maintained is $O(1)$ blocks. At the end of the file, the algorithm outputs the last $O(1)$ blocks of the encoding, which is now guaranteed to be prefix-free. The decoding algorithm has the same behavior. We can also decode or modify any block by reading/writing only $O(1)$ code blocks. In addition, we can append to the stream or delete blocks from the end in $O(1)$ time.

In Section 5, we consider the case of binary streams, and show how to match the optimal Elias bound up to $O(\log\log n)$:

Theorem 2. *There exists a universal code mapping n bits to $n + \log_2 n + O(\log\log n)$ bits, which can be encoded and decoded by algorithms that use $O(\log n)$ bits of space and take $O(1)$ time per operation.*

APPLICATIONS TO CRYPTOGRAPHY. As we already mentioned, prefix-free encodings (PFEs) are also essential for proving security of many natural cryptographic constructions which need to process variable-length messages. We will give many popular examples in Appendix A, but start with explaining the general reason why PFEs are useful in this context. In all such constructions, one first designs a "secure" *compression function* $f : \{0,1\}^s \times \{0,1\}^b \to \{0,1\}^s$, where b is called the block-length, and s is called the buffer-length. (Depending on the application, such f could be keyed or unkeyed, as we survey in Appendix A.) Intuitively, though, f allows one to securely process short, b-bit messages: given an initial buffer $IV \in \{0,1\}^s$ (which could be a key or a fixed constant), the hash of $m \in \{0,1\}^b$ is simply $y = f(IV, m)$. Then, to process an arbitrarily long message M, one first splits M into b-bits blocks $m_1 \dots m_n$ (so $n \approx |M|/b$),[2] and then iteratively process M via the *cascade mode-of-operation*, parameterized by the compression function f and the *initialization vector* $IV \in \{0,1\}^s$ (see Figure 1):

CASCADE(M, f, IV):

[2]We will assume that $|M|$ is a multiple of b. If not, one can always pad M with 1 followed by several (at most $(b-1)$) zeros to make the last block m_n full. Hence, w.l.o.g. we assume $M \in (\{0,1\}^b)^*$.

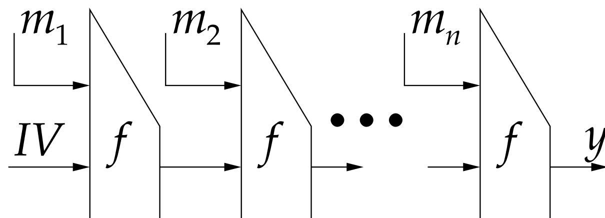

Figure 1: The Cascade Mode of Operation $\textsc{Cascade}(M, f, IV)$.

Split M into b-bit blocks $m_1 \dots m_n$, where
  $m_i \in \{0,1\}^b$ and $n = \lceil |M|/b \rceil$;
Set $x_0 = IV$;
For $i = 1$ to n
  Set $x_i = f(x_{i-1}, m_i)$;
Output $y = x_n$;

Typically, one can prove the following result: if the compression function f is "secure" *and M is encoded in prefix-free form*, then $\textsc{Cascade}(M, f, IV)$ is "secure" (where IV is either a cryptographic key or a constant, depending on the application). Intuitively, one can imagine building 2^b-ary tree T, whose root is labeled by the IV, and each internal node $m_1 \dots m_n$ at "depth" n is labeled by $\textsc{Cascade}(m_1 \dots m_n, f, IV)$. Typically, f is designed in a way that its ancestor-descendant node pairs might be related in a "meaningful way" (e.g., $\textsc{Cascade}(MM', f, IV) = \textsc{Cascade}(M', f, \textsc{Cascade}(M, f, IV))$), while all other pairs of nodes are not related. Thus, if no two messages M_1 and M_2 are in the ancestor-descendant relationship, then the construction is secure. But this precisely corresponds to the fact that the messages are encoded in a prefix-free way.

Moreover, in all known examples (see Appendix A), one can actually break the security of the construction if the PFE constraint is not enforced. Such attacks are typically referred to as *extension attacks*. Of course, if efficient one-line PFEs were known, one can easily prevent extension attacks using an on-line PFE. Unfortunately, since prior to our work it was believed (and explicitly conjectured by [MS05]) that one-line PFE comes at a significant cost, extension attacks are typically handled using various ad-hoc methods, different for each specific application. Specific examples are discussed in Appendix A, and include CBC-MAC [BRK94], cascade pseudorandom functions [BCK96a], Merkle-Damgård hash functions [CDMP05], domain extension of message authentication codes [MS05] and multi-property preservation [BR06, BR07]. In each of these examples, applying the cascade transform to the *prefix-free encoding* of the message gives the simplest-to-understand domain extension for the given cryptographic application. Thus, one application of our results is that previously used ad hoc methods might not be necessary in these applications.

1.3 Comparison to Arithmetic Coding

From an information-theoretic view point, the standard approach to converting from one base to another is *arithmetic coding*. To encode n independently chosen symbols from some distribution $\mathcal{D}$, arithmetic codes use $n \cdot \mathrm{H}(\mathcal{D}) + O(1)$ bits in expectation, where $\mathrm{H}(\cdot)$ denotes binary entropy. In our case, $\mathcal{D}$ is the uniform distribution over an alphabet Σ, so $n \log_2 \Sigma + O(1)$ bits are used. Unfortunately, a closer look at arithmetic coding reveals that it does not have the kind of worst-case locality properties that our applications require.

The basic idea in arithmetic coding is to map the space of possible inputs A to the interval $[0, 1]$. To represent the first letter, $[0, 1]$ is broken into $|\Sigma|$ intervals, and each letter $a \in \Sigma$ gets an interval of length $p_{\mathcal{D}}(a)$. The process continues recursively, subdividing the interval corresponding to $A[1]$ in

order to represent $A[2]$, etc. After the (tiny) interval corresponding to A has been identified, one simply outputs the shortest binary number inside it.

The challenge is to implement this conceptual idea efficiently with bounded precision arithmetic. The ubiquitous implementation, due to Witten et al. [WNC87, MNW98], can perform encoding and decoding of a stream in $O(1)$ time per bit, using $O(\log n)$ space.

It is instructive to briefly review this algorithm, as it highlights the inherently non-local nature of arithmetic codes. Imagine encoding a stream of symbols sequentially, and maintaining a subinterval of $[0,1]$ that represents the data so far. Whenever the interval is fully contained in $[0, \frac{1}{2}]$, a zero bit can be output, and we can re-normalize $[0, \frac{1}{2}] \mapsto [0,1]$ to gain precision. Similarly, once the interval is in $[\frac{1}{2}, 1]$, a one bit is output and we map $[\frac{1}{2}, 1] \mapsto [0,1]$. Alas, this is not enough to keep the required precision bounded, as the interval may become $[\frac{1}{2} - \varepsilon, \frac{1}{2} + \varepsilon]$ for exponentially small ε.

The trick of [WNC87, MNW98] is to re-normalize $[\frac{1}{4}, \frac{3}{4}] \mapsto [0,1]$ whenever the current interval is fully contained in $[\frac{1}{4}, \frac{3}{4}]$. Since the next bit is not known, the algorithm simply increments a counter of "outstanding bits." When the interval finally gets below or above the middle point, a burst of zeros or ones is written.

This demonstrates that the encoding of one particular character may depend non-trivially on many surrounding characters. It is thus impossible, for worst-case data, to access a symbol in the middle of the vector without linear decoding time. For prefix-free encoding/decoding, arithmetic coding would lead to bursty behavior where many blocks may be written after one block is read.

We remark that, if we could generalize our results by designing locally-decodable arithmetic codes for a biased distribution $\mathcal{D}$ (as opposed to the uniform distibution), one would immediately solve the dictionary problem (a.k.a. set membership) — perhaps the most important remaining open problem in this field.

1.4 Organization

In Section 2, we begin by describing a very simple algorithm for prefix-free encoding in the streaming model. This algorithm assumes that the input comes in blocks of $b \geq 2\log_2 n + 2$ bits, and outputs a code of at most $n+3$ blocks (or, $n+1$ blocks with a more complicated algorithm). While restrictive, this scenario is of interest in cryptography, where common message authentication schemes work with at least 128 bits at a time. The algorithm is very simple, and we expect it to be the practical choice for these application.

Using the intuition from this algorithm, we formulate a general concept (information carriers) in Section 3. Used appropriately, this construct implies an optimal data structure for representing a vector (Section 4), and an online prefix-free encoding of size $n + \log_2 n + O(\log\log n)$ (Section 5).

2 The SOLE Prefix-Free Encoding

2.1 The Basic SOLE

For the simplest algorithm, assume that the input stream consists of blocks of $b \geq 2\log_2 n + 1$ bits, and that n is odd (we can pad with an additional EOF block otherwise). Under these conditions, our algorithm will output an encoding of $n+2$ blocks. For each block, the algorithm uses $O(1)$ arithmetic operations on b-bit integers.

Let $B = 2^b$ be the alphabet of a block, and we write $[B]$ to denote the range $\{0, \ldots, B-1\}$. Adding a special end-of-file symbol (EOF), we obtain an alphabet of size $B+1$. Our goal is to encode $n+1$ letters from this alphabet (including the final EOF) using $n+2$ blocks of b bits.

The algorithm is best illustrated by Figure 2. Intuitively, it is simplest to view the algorithm as two separate passes through the stream.

Block Number	1	2	3	4	5	6	...	$n = 2i+1$	$2i+2$	$2i+3$
Input Alphabet	$[B]$	$[B]$	$[B]$	$[B]$	$[B]$	$[B]$	...	$[B]$	$\{\text{EOF}\}$	$\{0\}$
Size with EOF	$B+1$	$B+1$	$B+1$	$B+1$	$B+1$	$B+1$	...	$B+1$	$B+1$	$B-4i-4$
Pass 1	B	$B+4$	$B-4$	$B+8$	$B-8$	$B+12$	...	$B-4i$	$B+4i+4$	$B-4i-4$
Regroup	B	$B+4$	$B-4$	$B+8$	$B-8$	$B+12$	...	$B-4i$	$B+4i+4$	$B-4i-4$
Pass 2	B	B	B	B	B	B	...	B	B	B

Figure 2: Short-Odd Long-Even (SOLE) Encoding. Assumes n is odd and $B \geq 2n^2$.

In *Pass 1*, we consider pairs of elements on positions $(2i+1, 2i+2)$, for $i \geq 0$. Grouped together, the elements form a number in range $\left[(B+1)^2\right]$. We decompose this number into two values: one in range $\left[B-4i\right]$, and one in range $\left[B+4i+4\right]$. The smaller range is placed on odd positions, while the larger one on even positions, hence the name "Short-Odd Long-Even (SOLE)[3] Encoding."

The decomposition is simply a division by $B-4i$, using the remainder as the first block and the quotient as the next one. The transformation is valid as long as:

$$(B+1)^2 \leq (B-4i)(B+4i+4) \iff B \geq 2(2i+1)^2 - \frac{3}{2}$$

which follows from the fact that $2i+1 \leq n$ and $B \geq 2n^2$.

In *Pass 2*, we regroup the elements, considering pairs on positions $(2i, 2i+1)$ for $i \geq 1$. These elements come from range $\left[B+4i\right]$ and $\left[B-4i\right]$, respectively. Since $(B+4i)(B-4i) < B^2$, we can group them together as a number in range $[B^2]$. This uses a multiplication by $B+4i$. Hence, we have obtained $2b$ bits (a "double-block"), which we output immediately.

It is clear that these two conceptual passes can be implemented as a single pass in an online streaming algorithm. We simply need to remember the state of each pass (at most two integers each). The decoding algorithm is the straightforward inversion of this process, implementing Figure 2 bottom-up. One can immediately observe the following locality property:

Property 3. *Output blocks $2i$ and $2i+1$ can be computed from input blocks $2i-1, \ldots, 2i+2$. Input blocks $2i+1$ and $2i+2$ can be decoded from output blocks $2i, \ldots, 2i+3$.*

Termination. An important component of the algorithm that we have not described is the termination behavior, once EOF is received. We assumed that n was odd, i.e. EOF appears at some even position $n+1 = 2i$. The final element after Pass 1 is in range $\left[B+4i+4\right]$. We artificially insert a zero value in range $\left[B-4i-4\right]$ into the stream output by Pass 1. After regrouping and Pass 2, this completes a double-block at positions $2i+2 = n+1$ and $2i+3 = n+2$, which is output.

When the decoding algorithm has decoded the EOF symbol, it stops immediately. Thus, we need to argue that the decoding stops before reading past the end of the encoded file. This follows from Property 3, as the last block needed in the decoding of EOF is $2i+1$.

2.2 Small Example

Assume the number of blocks $n = 3$ and $B = 32$, so each block is a 5-bit number in the range $[32] = \{0, \ldots, 31\}$ and the last index $i = 1$. This is OK as n is odd and $32 > 2 \cdot 3^2 = 18$. Figure 3 denotes the specialization of Figure 2 to this concrete setting. In contrast, Figure 4 denotes the actual run of the encoding (and recoding, if done in reverse) on 3-block input $(5, 7, 31)$. Please refer to these two Figures to follow the calculations below.

[3] Coincidentally, one meaning of the word "sole" is "exclusive, not shared with others".

Block Number	1	2	3	4	5
Input Alphabet	[32]	[32]	[32]	{EOF}	{0}
Size with EOF	33	33	33	33	24
Pass 1	32	36	28	40	24
Regroup	32	36	28	40	24
Pass 2	32	32	32	32	32

Figure 3: SOLE Encoding for $n = 3$ and $B = 32$. Note how, for example, $36 \cdot 28 < 32^2 < 33^2 < 28 \cdot 40$.

Block Number	1	2	3	4	5
Input Alphabet	[32]	[32]	[32]	{EOF}	{0}
Actual Input	$5 \in [32]$	$7 \in [32]$	$31 \in [32]$	EOF	n/a
With EOF	$5 \in [33]$	$7 \in [33]$	$31 \in [33]$	$32 \in [33]$	$0 \in [24]$
Pass 1	$12 \in [32]$	$7 \in [36]$	$23 \in [28]$	$38 \in [40]$	$0 \in [24]$
Regroup	$12 \in [32]$	$7 \in [36]$	$23 \in [28]$	$38 \in [40]$	$0 \in [24]$
Pass 2	$12 \in [32]$	$3 \in [32]$	$26 \in [32]$	$6 \in [32]$	$1 \in [32]$

Figure 4: Actual run for $n = 3$, $B = 32$ and $M = (5, 7, 31)$. Output is $E(M) = (12, 3, 26, 6, 1)$.

2.2.1 Encoding

Say, the 3 input blocks are $5, 7, 31$. First, we attach the EOF, which is encoded as 32. We also add the last 0 which is viewed as being in range $B - 4i - 4 = 32 - 4 - 4 = 24$. Thus, we get a modified stream of 5 blocks $5, 7, 31, 32, 0$, where $5, 7, 31, 32 \in [33]$ and $0 \in [24]$. We get two "double blocks" $(5, 7)$ and $(12, 31)$. Let us process them one-by-one, as a streaming algorithm would do.

First Double Block. We need to encode $(5, 7) \in [33] \times [33]$ as two numbers in the range $[32] \times [36]$. First, from $x = 5$ and $y = 7$, we obtain a large number $z = 7 \cdot 33 + 5 = 236$. Then we uniquely decompose 236 modulo 32: $236 = 7 \cdot 32 + 12$. Thus, our first double block is $(12, 7) \in [32] \times [36]$.

Second Double Block. We need to encode $(31, 32) \in [33] \times [33]$ as two numbers in the range $[28] \times [40]$. First, from $x = 31$ and $y = 32$, we obtain a large number $z = 32 \cdot 33 + 31 = 1087$. Luckily, as we expected, $1087 < 28 \cdot 40 = 1120$, so we can uniquely decompose 1087 modulo 28 as: $1087 = 38 \cdot 28 + 23$. Thus, our second double block is $(23, 38) \in [28] \times [40]$.

Output of Pass 1 and Regrouping. Thus, after Pass 1, we transformed our modified stream $(5, 7, 31, 32, 0) \in [33] \times [33] \times [33] \times [33] \times [24]$ into $(12, 7, 23, 38, 0) \in [32] \times [36] \times [28] \times [40] \times [24]$. We now regroup it into a singleton element $12 \in [32]$, and two double blocks $(7, 23) \in [36] \times [28]$ and $(38, 0) \in [40] \times [24]$, and we are ready for Pass 2.

Output of Pass 2. The first singleton $12 \in [32]$ we output immediately, as it fits into 5 bits already. Next, we transform the first double blocks $(7, 23) \in [36] \times [28]$ into a 10-bit number in range $[32^2] = [1024]$ by setting this number to $23 \cdot 36 + 7 = 835$. We can write it as 10 bits, but for better understanding, let's convert it into two numbers in $[B] = [32]$ by decomposing $835 = 26 \cdot 32 + 3$, so we get a double block $(3, 26) \in [32] \times [32]$. We immediately output these numbers.

Similarly, we transform the second (and last) double blocks $(38, 0) \in [40] \times [24]$ into a 10-bit number in range $[32^2] = [1024]$ by setting this number to $0 \cdot 40 + 38 = 38$.[4] We can write it as 10 bits (in fact, 6 bits, see the Footnote 4), but for better understanding, let's convert it into two numbers in $[B] = [32]$. We do it by decomposing $38 = 1 \cdot 32 + 6$, getting a double block $(6, 1) \in [32] \times [32]$, which we immediately output.[5]

[4]Since we always append 0 as the last value in the range $[B - 4i - 4]$, the output is always the previous block whose actual range $[B + 4i + 4]$ is only slightly larger than $[B]$. In particular, we need only $b + 1$, and not $2b$, bits for this number. See Section 2.3 for a more tighter variant of SOLE which uses this observation.

[5]Once again, notice that the last block is always either 0 or 1, depending if the converted number was less than B or

Final Output. To summarize, we transform three input blocks $5, 7, 31$ into 5 output blocks $12, 3, 26, 6, 1$.

2.2.2 Decoding

We now show how to decode five blocks $12, 3, 26, 6, 1$ back to $5, 7, 31$.

Recovering Output of Pass 2. We first need to recover the output of Pass 2. The first block 12 is already in Pass 2. We now take the next double block $(3, 26) \in [32] \times [32]$, and convert it into double block in the range $[36] \times [28]$. First, we write our block as one big number $26 \cdot 32 + 3 = 835$. We then decompose $835 = 23 \cdot 36 + 7$, which indeed gives us the correct block $(7, 23) \in [36] \times [28]$.

Similarly, we transform the second (and last) double blocks $(6, 1) \in [32] \times [32]$ into a block in range $[40] \times [24]$ by first recovering the "large" number $1 \cdot 32 + 6 = 38$, and then decomposing it as $38 = 0 \cdot 40 + 38$. This gives us the next double block $(38, 0)$. Notice, we actually *do not know yet this is the last block*, but let us move to the decoding of Pass 1, which happens in parallel, and which will let us figure this out!

Recovering Output of Pass 1. As we just saw, we recovered the output $12, 7, 23, 38, 0$ of Pass 1, although we do not know that this is the end yet. How do we find out? Because we actually try to recover the input to Pass 1, as we recover the output of Pass 2 (which is also the output of Pass 1 re-grouped). Let's do it!

After regrouping, we get double blocks $(12, 7) \in [32] \times [36]$, $(23, 38) \in [28] \times [40]$, and not yet (and never!) completed element 0 (which is the last 0, but we do not know it yet). We transform the first block $(12, 7)$ into a large number $7 \cdot 32 + 12 = 236$, and then write $236 = 7 \cdot 33 + 5 = 236$. This gives us the first input block $(5, 7)$, which we immediately output, since none of the numbers is equal to $32 = \text{EOF}$, so we know this is not the end of the file.

We then do the same thing for the second block $(23, 38) \in [28] \times [40]$, transforming it into a large number $38 \cdot 28 + 23 = 1087$, which we then write as $1087 = 32 \cdot 33 + 31$. This gives us the next input block $(31, 32)$. So we output 31 and *stop*, since we just recovered the next symbol 32, which is end-of-file.

Thus, we indeed recovered three blocks $5, 7, 31$, and do know there is no more data coming.

2.3 Generalizations and Further Discussion

Additional properties. Based on Property 3, one can support random access to the encoding in constant time. Decoding a block in the middle of the file requires reading 4 output blocks. Modifying an input block will read and change 4 consecutive output blocks. For instance, to modify block 3 in Figure 2, we first read output blocks $2, \ldots, 5$. From these, we compute the input block 4, and blocks 2 and 5 output by Pass 2. From the new value of block 3 and the old block 4, we can rerun Pass 1 to update the intermediate blocks 3 and 4. We now know the intermediate blocks $2, \ldots, 5$, so the output blocks can be computed by running Pass 2. Appending to the file and truncating can be reduced to write operations.

Practical considerations. To ensure fast arithmetic operations, one would set $b = 32$ or $b = 64$. While the algorithm uses arithmetic on double precision ($2b$ bits), it is standard to implement division and multiplication by $2^b \pm x$ using fast, single-precision operations.

For a given b, our basic algorithm can process a stream of up to $n_0 = 2^{(b-1)/2}$ blocks. For $n > n_0$ blocks, we can trivially obtain an encoding with overhead $\frac{n}{n_0} + O(1)$ blocks, by applying SOLE on each chunk of n_0 blocks. Specifically, one block is wasted for each chunk except the last one, where 3 blocks may be wasted.

between B and $B + 4i + 3$.

Block Number	1	2	3	4	5	6	...
Input Alphabet	$[B]$	$[B]$	$[B]$	$[B]$	$[B]$	$[B]$	...
Size with EOF	$B+1$	$B+1$	$B+1$	$B+1$	$B+1$	$B+1$	...
Pass 1	$B+1$	$B-1$	$B+5$	$B-5$	$B+9$	$B-9$	...
Regroup	$B+1$	$B-1$	$B+5$	$B-5$	$B+9$	$B-9$	...
Pass 2	B	B	B	B	B	B	...

Figure 5: Long-Odd Short-Even (LOSE) Encoding.

With a minimal setting of $b = 32$ bits, our encoding adds a block (4 bytes) per each $2^{15.5} = 46,340$ blocks (≈ 181Kb), making our overhead roughly $2^{-15.5} \approx 0.002\%$. For comparison, the naïve encoding will waste 1 bit per block, which is $\frac{1}{32} = 3.125\%$, a factor of 1448 worse than our encoding. E.g., a 32Gb file will have a negligible 707Kb overhead with our encoding, and 1Gb overhead with the naïve encoding.

A TIGHTER ENCODING. The previous algorithm wasted up to 3 blocks (when n is even). If $n \geq 2$ and $b \geq 2\log_2 n + 2$, we can instead obtain an optimal encoding that always uses $n+1$ output blocks. The first idea is to conceptually insert the EOF symbol two blocks before the actual termination of the input stream. This can be done by buffering the last two blocks, and ensures that EOF will be followed by two output blocks. Hence, by Property 3, the usual decoding algorithm will certainly observe the EOF without reading past the end of the encoding. At this point, we switch to a special termination procedure for the last two blocks.

Imagine that, after EOF appearing on position $n-1$, and the last two blocks on positions n and $n+1$, an infinite stream of zeros follows in the input stream. Then, it is not hard to see that block $n+2$ of the usual encoding will be 0 or 1 (henceforth “the final bit”; see the example in Section 2.2 for an illustation of this observation), and the remaining blocks are guaranteed to be zero. In the new algorithm, we will output the usual output values of blocks up to $n+1$.

Instead of wasting block $n+2$ for the final bit, we will use the following hack: we will have two end-of-file symbols, EOF_0 and EOF_1, coding this bit. We note that there is no circular dependence, since block $n+2$ (the final bit) does not depend on block $n-1$ (the EOF) by Property 3. The price we pay is increasing the alphabet to $B+2$, instead of $B+1$. Now ranges of the form $[B \pm 4i]$ become $[B \pm 8i]$. The encoding is possible as long as:

$$(B+2)^2 \leq (B-8i)(B+8i+8) \iff B \geq 4(2i+1)^2 - 3$$

Since $2i+1 \leq n$, this requires $b \geq 2\log_2 n + 2$.

This scheme is further improved in Section 5, which also uses multiple EOF symbols, and moves EOF ahead of the file termination.

LOSE ENCODINGS. An alternative scheme with almost identical properties is the Long-Odd Short-Even (LOSE) encoding; see Figure 5 (for simplicitly, we did not describe the termination policy of LOSE in the Figure 5, but it is similar to that of SOLE).

3 Information Carriers

Let us re-examine the behavior of SOLE encodings for intuition. After seeing $2n$ blocks, the algorithm has written $2n-1$ blocks to the output, and is left with a number in range $\lceil B+4n \rceil$. In keeping with the terminology of [Păt08], we shall call this number a *spill*.

The algorithm proceeds by grabbing two more blocks, i.e. a number in $\lceil (B+1)^2 \rceil$. Its first worry is to *complete* the spill with some information, to obtain two full blocks of output. That means that the spill must be combined with a value in range $\lfloor \frac{B^2}{B+4n} \rfloor \approx B - 4n$.

To identify some useful information in range $\big[B-4n\big]$, the algorithm takes the new number in $\big[(B+1)^2\big]$ modulo $B-4n$. The result is combined with the spill and immediately output. Remaining now is the quotient of the division, a number in range $\frac{(B+1)^2}{B-4n} < B+4n+4$. This is the new spill.

More abstractly, imagine the current spill as a value y in some range $[Y]$. We would like to store y immediately, but, unfortunately, the range $[Y]$ is not a power of two, so we cannot simply write y to memory without losing entropy. Instead, we use some unrelated x in some appropriately bigger range $[X]$ as an "information carrier." The purpose of x is to help commit y to memory without much waste. The pair (x, y) is injectively mapped into the immediate output m and the next spill s, such that:

- m comes from some range $[2^M]$, meaning that it can be written to memory *in bits*, without losing entropy;
- m stores enough information to recover the old spill y we care about, meaning that y can be immediately recovered from memory.
- the entropy loss of going from (x, y) to (m, s) can be made arbitrarily small, by making the range $[X]$ of the information carrier appropriately large.

For instance, in the SOLE encoding, we had $Y = B + 4n$. By making $X = (B + 1)^2$, we could store the spill y in $M = 2b$ bits of memory, and the new spill s had only a marginally larger range $[S] = [B+4n+4]$. This means that the entropy loss of this operation was at most $\log_2(B+4n+4) - \log_2(B + 4n) = O(1/B)$ bits.

The discussion above motivates us to formulate the following general lemma, where we optimized the parameters M and S to minimize the entropy loss:

Lemma 4. *Let $X, Y \le 2^w$. Then, for some values M and S, we can injectively map $(x, y) \in [X] \times [Y]$ to $(m, s) \in [2^M] \times [S]$, with the following properties:*

- *S and M are chosen as functions of X and Y, but satisfy $S = O(\sqrt{X})$ and $2^M = O(Y\sqrt{X})$;*
- *the map can be evaluated in $O(1)$ time;*
- *x can be decoded from m and s, in $O(1)$ time;*
- *y can be decoded from m alone, in $O(1)$ time;*
- *the redundancy of this encoding scheme is:*

$$\big(M + \log_2 S\big) - \big(\log_2 X + \log_2 Y\big) \; = \; O\left(\tfrac{1}{\sqrt{X}}\right).$$

The lemma could be graphically depicted by Figure 6. The current spill value y is combined with the information carrier x, producing the memory content m (from which one can recover y) and the new spill value s.

Proof. Say we will use M memory bits. The entire information about y must be in these M bits. In addition to y, the bits can store a quantity from the universe $C = \lfloor 2^M/Y \rfloor$. The redundancy in doing this rounding is at most:

$$\begin{aligned} R_1 \quad &= \quad M - \log_2\left(Y \cdot \left\lfloor \frac{2^M}{Y} \right\rfloor\right) \; \le \; M - \log_2\left(2^M - Y\right) \\ &= \quad \log_2\left(\frac{1}{1 - Y/2^M}\right) \; = \; O\left(\frac{Y}{2^M}\right) = O\left(\frac{1}{C}\right). \end{aligned}$$

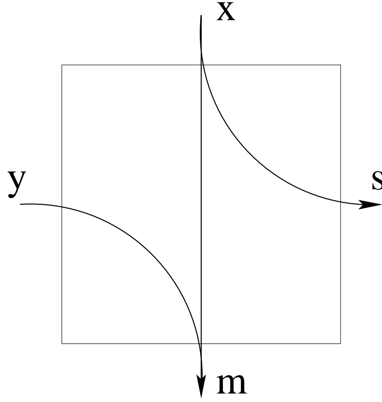

Figure 6: Lemma 4 implements the box depicted.

Now x is broken into two components: a value in the universe $[C]$, to be combined with y, and a spill of size $S = \lceil X/C \rceil$. This decomposition introduces a redundancy of:

$$\begin{aligned} R_2 &= \left(\log_2 C + \log_2 \left\lceil \frac{X}{C} \right\rceil\right) - \log_2 X \\ &\leq \log_2 \left(\frac{X+C}{X}\right) \leq O\left(\frac{C}{X}\right). \end{aligned}$$

Balancing the two redundancies, $R_1 = R_2$, we find that we should set $C = \Theta(\sqrt{X})$ and obtain $R_1 = R_2 = O(\frac{1}{\sqrt{X}})$. The overall redundancy is, by the chain rule:

$$\begin{aligned} &\big(M + \log_2 S\big) - \big(\log_2 X + \log_2 Y\big) \\ &\quad= \big(M - \log_2 Y - \log_2 C\big) + \big(\log_2 C + \log_2 S - \log_2 X\big) \\ &\quad= R_1 + R_2. \end{aligned}$$

Mapping (x, y) to (m, s) takes one division and one multiplication. (We assume here that X and Y are fixed, so constant like C are precomputed.) Extracting y from m takes one division, and extracting x from (m, s) takes one division and one multiplication. In practice, it is standard to replace any integer division by a constant with two multiplications (precomputing the multiplicative inverse to w bits of precision). ■

4 Representing a Vector

4.1 A First Attempt

We now turn to the problem of representing a vector $A[1 \,..\, n]$ of elements from Σ, supporting local access. A direct approach is to simply iterate Lemma 4. Formally, we group elements of A into blocks of size $O(\log_\Sigma n)$, so that each block can be interpreted as an element x in range $[X]$, where $X = \Theta(n^2)$. This gives us $N = O(n/\log_\Sigma n)$ larger elements $x_1, \ldots, x_N$ from a universe of size $\Theta(n^2)$. Initially, we start with a null spill, $Y_0 = 0$. Applying the lemma, we obtain some memory bits M_0 (which we write to memory), and some spill Y_1. Then, we store the spill from Y_1 by using x_2 as the information carrier, and obtain a spill Y_2, etc. Observe that, while the spill size Y_i may depend non-trivially on Y_{i-1}, our lemma guarantees that $Y_i = O(\sqrt{X})$, i.e. the spill universe doesn't grow without bounds.

The redundancy introduced in each step is $O(1/\sqrt{X}) = O(1/n)$. Thus, the overall redundancy is $O(\frac{N}{n}) = o(1)$, plus one bit at the end, when we simply write down the last spill. To prove this formally, note that our definition of redundancy admits a chain rule:

$$\begin{aligned}
&\big(\lceil \log_2 Y_n \rceil + \sum_i M_i\big) - n \log_2 \Sigma \\
&\le 1 + \sum_i \big(M_i + \log_2 Y_i - \log_2 Y_{i-1}\big) - N \log_2 X \\
&= 1 + \sum_i \big(M_i + \log_2 Y_i - \log_2 Y_{i-1} - \log_2 X\big) \\
&\le\ 1 + N \cdot O\left(\tfrac{1}{\sqrt{X}}\right)
\end{aligned}$$

To read an element $A[i]$, we first calculate to which x_j it belongs. Then, we recover the spill y_j from the memory written in step $j+1$. Finally, we recover x_j from the spill y_j and the memory written in step j. Updating an x_j can be done by recomputing the actions of steps j and $j+1$. Spill y_j may change in the process, but this only affect the memory written in step $j+1$, and does not propagate to y_{j+1}.

Unfortunately, this approach requires $N = O(n/\log n)$ precomputed constants. Indeed, each Y_j is distinct, and even the number of bits written to memory in each step varies. This was not a problem in the SOLE encoding, as $X = 2^b + 1$, so the Y_j values could be approximated by an arithmetic progression. Here, X can be far from a power of two, and the behavior of the Y_j's can be rather irregular inside a universe of $O(n)$.

4.2 A Tree Representation

To reduce the number of precomputed constants to $O(\log n)$, we will use a tree structure. As before, we group elements into N values from universe $X = \Theta(n^2)$. Group these values into a binary tree, as in the standard representation of a heap. This is an implicit representation, i.e. we simply *define* $2i$ and $2i+1$ as the children of node i, and $\lfloor i/2 \rfloor$ as the parent of node i.

Each node will receive two spills, one from each child. It will combine these into a single value y_i, and use its own value $x_i \in [X]$ as the information carrier to store y_i. The resulting new spill is propagated to the parent. Note that, even though spills are propagated up, there is no recursion happening at query time. To obtain a value stored in a node v, we go to v's parent and obtain v's spill from the parent's memory bits. Then, we compute v's value from the spill and its own memory bits. To update some x_i, we simply reconstruct the encoding of node i and its parent.

The advantage of the arboral representation is that all nodes on a level are handled using $O(1)$ precomputed constants. We have three types of nodes on level $\lceil \log_2 n \rceil - k$:

- A prefix of the nodes have subtrees that are complete binary trees of height k.
- Following this, at most one node has a subtree that is not complete.
- The remaining nodes have subtrees that are complete binary trees of height $k-1$.

The nodes in each of these three classes, from a given level, have identical constants generated by Lemma 4, since the spills from the children are identical by induction. Therefore, for each class of nodes on each level, we need to remember the following information:

- the number of nodes in this class;
- the location of the memory bits from the first node in the class;

- the spill sizes received from the two children;
- the constants of Lemma 4.

The level of node i is $\lfloor \log_2 i \rfloor$. We note that this can be computed by $O(1)$ Word RAM operations, as shown in [FW93]. Once the level is known, we can determine in constant time the class of the node, and retrieve the constants and memory locations necessary for decoding.

THE LAST BIT. By the construction from above, the total redundancy introduced by the applications of Lemma 4 is $o(1)$ bits. Up to one bit of redundancy is then lost by rounding when the final spill is stored. Thus, we use $\lceil n \log_2 \Sigma + o(1) \rceil$, which could be equal to $\lceil n \log_2 \Sigma \rceil + 1$.

As a theoretical amusement, we mention that the space can be reduced to the optimal $\lceil n \log_2 \Sigma \rceil + 1$. The idea is that the $o(1)$ term can be made $O(n^{-c})$, for any constant c, if one increases X and uses arithmetic on $O(c \log n)$-bit integers. But results on linear forms in logarithms show [BW92] that $n \log_2 \Sigma$ is bounded away from an integer by $1/n^{O(1)}$, for any constant Σ. Thus, setting c high enough will ensure that $n \log_2 \Sigma + O(1/n^c) \leq \lceil n \log_2 \Sigma \rceil$. (For instance, when $\Sigma = \{0, 1, 2\}$, it suffices to set $c = 1,179,648$.)

5 Prefix-Free Codes via Information Carriers

In this section, we show how to obtain online prefix-free codes of size $n + \log_2 n + O(\log \log n)$, getting exponentially closer to the Elias bound of $n + \log_2 n + \log_2 \log_2 n + \dots$

We view the input as partitioned into blocks of slowly growing size: after n bits have been seen, the next block (block i) consists of the subsequent $2k_i = 2\lceil \log_2 n \rceil$ bits. Let N be the number of blocks. As the N-th block may contain anywhere between 1 and $2\lceil \log_2 n \rceil$ bits, we treat it specially. We will encode the first $N - 1$ blocks in a prefix-free way, and then append an Elias encoding for the last block. This Elias code brings an overhead of $O(\log k_N) = O(\log \log n)$ bits.

We view a block as coming from the alphabet $[2^{k_i}]^2$. The reason for blocks of around $2 \log_2 n$ bits is the information carrier lemma: when working with a domain of X, we obtain a redundancy of $O(1/\sqrt{X})$. Thus, a block universe of $O(n^2)$ is needed to control the redundancy introduced in each block.

Unfortunately, we cannot afford one block for the EOF symbol, since this would cost some $2 \log_2 n$ bits. Instead, in Pass 1 we convert each block of $[2^{k_i}]^2$ to a block of $[2^{k_i}] \times \Big([2^{k_i}] \cup \{\text{EOF}\} \Big)$. More precisely, we use a buffer of $O(\log n)$ bits to introduce a 3-block lag, effectively learning about the end of file 3 blocks in advance. If block $N - 2$ has value $(a, b) \in [2^{k_{N-2}}]^2$, we replace it by (a, EOF), and append b at the very end of the stream (in plaintext). This transformation costs $k_{N-2} \leq \log_2 n$ bits for the EOF symbol. In addition, we pay $\log_2 \left(2^{k_i} (2^{k_i} + 1) \right) - \log_2 \left(2^{2k_i} \right) = O(2^{-k_i})$ for enlarging the alphabet of block i. To understand $\sum_i 2^{-k_i}$, note that there are $O(2^K / K)$ blocks with a fixed value $k_i = K$, so the total cost from fixed K is $O(1/K)$. Summing for K from 1 to $\log_2 n$, we obtain $O(\log \log n)$ bits of redundancy.

In Pass 2, we convert each block from alphabet $[2^{k_i}] \times \Big([2^{k_i}] \cup \{\text{EOF}\} \Big)$ to binary. This proceeds like in Section 4.1, by applying Lemma 4 sequentially to all blocks, and feeding one spill into the next application. The redundancy introduced at step i is $O(2^{-k_i})$, since the domain is $X = O(2^{2k_i})$. As shown above, $\sum_i 2^{-k_i} = O(\log \log n)$. For the decoding, observe that EOF was encoded in block $N - 2$, so it will be detected after output block $N - 1$ is read (which allows one to decode the spill of input block $N - 2$, and thus the entire $N - 2$ block). The decoder can thus stop in time at block $N - 1$, and run the special decoding for block N, and the second half of block $N - 2$ displaced by EOF.

Overall, the overhead was $\log_2 n + O(\log \log n)$, and the running time per block was constant.

References

[AB99] Jee Hea An, Mihir Bellare, *Constructing VIL-MACs from FIL-MACs: Message Authentication under Weakened Assumptions*, CRYPTO 1999, pages 252-269. 17

[BW92] Alan Baker and Gisbert Wüstholz. Logarithmic forms and group varieties. *Journal für die reine und angewandte Mathematik*, 442:19–62, 1992. 13

[Bel06] Mihir Bellare, *New Proofs for NMAC and HMAC: Security without Collision-Resistance*, Advances in Cryptology - Crypto 2006 Proceedings, Springer-Verlag, 2006. 16

[BCK96a] Mihir Bellare, Ran Canetti, and Hugo Krawczyk, *Pseudorandom Functions Re-visited: The Cascade Construction and Its Concrete Security*, In Proc. 37th FOCS, pages 514-523. IEEE, 1996. 2, 4, 16

[BCK96b] Mihir Bellare, Ran Canetti, Hugo Krawczyk: *Keying Hash Functions for Message Authentication*. CRYPTO 1996: pp. 1–15. 16

[BRK94] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The Security of Cipher Block Chaining. In *Crypto '94*, pages 341–358, 1994. LNCS No. 839. 2, 4, 16

[BR06] Mihir Bellare and Thomas Ristenpart, *Multi-Property-Preserving Hash Domain Extension and the EMD Transform*, In Advances in Cryptology - Asiacrypt 2006, pp. 299-314. 4, 18

[BR07] Mihir Bellare and Thomas Ristenpart, *Hash Functions in the Dedicated-Key Setting: Design Choices and MPP Transforms*, ICALP 2007, pp. 399-410. 4, 17, 18

[BR93] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *Proceedings ACM Conference on Computer and Communications Security* (1993), 62-73. 16

[CDMP05] Jean-Sebastian Coron, Yevgeniy Dodis, Cecile Malinaud and Prashant Puniya, *Merkle-Damgård Revisited: How to Construct a Hash Function*, Advances in Cryptology, Crypto 2005 Proceedings: 430-448, Springer-Verlag, 2006. 2, 4, 17

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993. See also STOC'90. 13

[GGG+07] Alexander Golynski, Roberto Grossi, Ankur Gupta, Rajeev Raman, and S. Srinivasa Rao. On the size of succinct indices. In *Proc. 15th European Symposium on Algorithms (ESA)*, pages 371–382, 2007. 1

[GM03] Anna Gál and Peter Bro Miltersen. The cell probe complexity of succinct data structures. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 332–344, 2003. 1

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986. 2, 16

[Gol09] Alexander Golynski. Cell probe lower bounds for succinct data structures. In *Proc. 20th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 625–634, 2009. 1

[HRSD07] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 389–400, 2007. 2

[Jac89] Guy Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989. 1

[Mit02] Michael Mitzenmacher. Compressed Bloom filters. *IEEE/ACM Transactions on Networking*, 10(5):604–612, 2002. See also PODC'01. 2

[MNW98] Alistair Moffat, Radford M. Neal, and Ian H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems (TOIS)*, 16(3):256–294, 1998. See also DCC'95. 5

[MS05] Ueli M. Maurer and Johan Sjödin. Single-key AIL-MACs from any FIL-MAC. In *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, pages 472–484, 2005. 2, 3, 4, 17

[Păt08] Mihai Pătraşcu. Succincter. In *Proc. 49th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008. 1, 2, 9

[PV10] Mihai Pătraşcu and Emanuele Viola. Cell-probe lower bounds for succinct partial sums. In *Proc. 21st ACM/SIAM Symposium on Discrete Algorithms (SODA)*, 2010. 1

[PR00] Erez Petrank, Charles Rackoff, *CBC MAC for Real-Time Data Sources*, J. Cryptology 13(3): 315-338 (2000). 16

[RS06] Vijayshankar Raman and Garret Swart. How to wring a table dry: Entropy compression of relations and querying of compressed relations. In *Proc. 32nd International Conference on Very Large Data Bases (VLDB)*, pages 858–869, 2006. 2

[Vio09] Emanuele Viola. Bit-probe lower bounds for succinct data structures. In *Proc. 41st ACM Symposium on Theory of Computing (STOC)*, pages 475–482, 2009. 2

[WNC87] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987. 5

A Applications to Cryptography

We now discuss several applications of prefix-free encodings (PFEs) in cryptography. As discussed in the the Introduction, all these applications could be viewed as processing the given n-block message $M = (m_1, \ldots, m_n)$ by the *cascade mode of operation*, parameterized by the appropriate *compression function* $f : \{0,1\}^s \times \{0,1\}^b \to \{0,1\}^s$ and the *initialization vector* $IV \in \{0,1\}^s$. (Depending on the application, such f could be keyed or unkeyed, and the IV could be a key or a constant.) For completeness, we repeat this definition of the cascade mode below (see also Figure 1):

Cascade(M, f, IV):
  Split M into b-bit blocks $m_1 \ldots m_n$, where
    $m_i \in \{0,1\}^b$ and $n = \lceil |M|/b \rceil$;
  Set $x_0 = IV$;
  For $i = 1$ to n
    Set $x_i = f(x_{i-1}, m_i)$;
  Output $y = x_n$;

CBC-MAC. The CBC-MAC is a very popular way to build a message authentication code (MAC) from a block cipher. Such block-cipher g can be viewed as a keyed permutation $g_k(m)$, where $|m| = b$,

k is the secret key and m is the message. The CBC-MAC of M is defined by $\textsc{Cascade}(M, f_k, 0^b)$, where the compression function $f_k(x, m) = g_k(x \oplus m)$ is keyed and the buffer-length $s = b$. Namely,

$$\text{CBC-MAC}(k, M) = g_k(m_n \oplus g_k(m_{n-1} \oplus \ldots \oplus g_k(m_2 \oplus g_k(m_1)) \ldots))$$

It is well known that CBC-MAC is secure if the encoding M is prefix-free [BRK94]. In contrast, CBC-MAC is easy to break otherwise. For example, if $y = \text{CBC-MAC}(k, m_1) = g_k(m_1)$, then setting $m_2 = m_1 \oplus y$, we get $\text{CBC-MAC}(k, m_1 m_2) = g_k(m_2 \oplus y) = g_k(m_1) = y$, meaning that it is easy to forge the MAC of (m_1, m_2) after learning the MAC of m_1.

The practical fix for the above problem (see [PR00]) is to use "encrypted CBC-MAC": namely, have a fresh key k' in addition to k, and output $g_{k'}(\text{CBC-MAC}(k, M))$. Although this fix is quite simple, it doubles the size of the secret key, which could be undesirable for some applications. On the other hand, our method makes at most one more calls to the block cipher (due to adding two more blocks in our encoding), and also adds a little bit of complexity due to encoding and decoding procedure (which are simple, but nevertheless need to be done).

Cascade PRF. Another natural application of the cascade mode is to extend the domain of a pseudorandom function (PRF). Specifically, assume g_k is a keyed PRF from b bits to s bits, where s is simultaneously the size of the key k and the output length of $g_k(m)$. Define the compression function $f(x, m) = g_x(m)$, and set the initial $IV = k$. Then the classical cascade construction is defined as $F_k(M) = \textsc{Cascade}(M, f, k)$. Namely, the cascade state x_i keeps the current PRF key (initially equal to the actual key k), with each iteration using the next message block m_i to set the new key x_i to the the PRF of the message block m_i under the old key x_{i-1}. We also remark that the classical GGM [GGM86] construction of a PRF F_k from a length-doubling pseudorandom generator (PRG) $G : \{0,1\}^s \to \{0,1\}^{2s}$ is also a special case of the cascade construction with $b = 1$. Indeed, the PRG $G(k) = (G_0(k), G_1(k))$ above, where $|G_m(k)| = s$ and $m \in \{0,1\}$, yields a 1-bit PRF $g_k(m) = G_m(k)$.

It is well known [GGM86, BCK96a] that the cascade construction is a secure PRF of variable-length messages if and only if the messages are encoded in a prefix-free form. Intuitively, as long as any two message "diverge" at some point, the resulting key values become random and unrelated. In contrast, the extension attack is particularly simple and devastating. If $M_2 = M_1 M'$, then $F_k(M_2) = F_{F_k(M_1)}(M')$, so one can easily compute the PRF of any "descendant" (M_1, M_2) from the PRF of the "ancestor" M_1, breaking the pseudorandomness of F_k.

The clean fix is to use the NMAC variant [BCK96b, Bel06], where a fresh key k' is stored in addition to k, and one defines $\text{NMAC}_{k,k'}(M) = g_{k'}(G_k(M))$. Once again, although simple, this doubles the key size of our PRF. Hence, in practice an hoc variant of NMAC is used, called HMAC [BCK96b, Bel06]. HMAC uses a single key $\tilde{k}$, and essentially calls NMAC with $k = g_c(\tilde{k} + ipad)$ and $k' = g_c(\tilde{k} + opad)$, where $c, ipad, opad$ are specific constants hardwired into the design of HMAC. This regains the short key size, but now the security analysis requires an ad hoc assumption about our initial PRF g, beyond the fact that it is a PRF. In contrast, using a PFE, one naturally regains a singe key k, has the same number of calls to g_k, but adds a little bit of complexity due to on-line encoding/decoding, which is not needed in HMAC.

Merkle-Damgård Hash Functions. The cascade mode of operation is the main technique for building cryptographic hash functions. Such a hash function H should take an arbitrary long message $M = (m_1, \ldots, m_n)$, and produce a short, s-bit hash value $H(M)$. The minimal requirement for such hash functions is collision-resistance: it should be hard to find $M_1 \neq M_2$ such that $H(M_1) = H(M_2)$. However, in many applications even stronger properties are needed from H: intuitively, a new value $H(M)$ should look random and independent from all previous values $H(M')$ for $M' \neq M$. This is formalized by modeling H as a *random oracle* [BR93]. Of course, in practice one uses concrete hash function H. In fact, practical hash functions, including the current standard SHA and the vast majority of other popular hash functions, precisely use the cascade mode of operations, applied to an

appropriate compression function f.[6]

Specifically, $H(M)$ is precisely $\textsc{Cascade}(M, f, IV)$, where f is a publicly known unkeyed hash function from $(s+b)$ to s bits, and IV is a fixed constant. As observed by many papers, and formally addressed by Coron et al. [CDMP05], such design of H makes it very different from a "monolithic" random oracle, even if the compression function f is modeled as a fixed-length random oracle. In particular, the extension attack here is as follows: given $y = H(M)$ for any *unknown* value M, one can easily compute the value $y' = H(M, M') = \textsc{Cascade}(M', f, y)$, for any suffix M'. Clearly, this should not be possible if H was a true random oracle.[7]

Coron et al. [CDMP05] formally defined what it means to securely extend the domain of a random oracle, and gave four simple ways to *provably* circumvent extension (and all other generic) attacks. The cleanest solution is to simply encode the message in a PFE! However, prior to our work it was considered too wasteful. The second solution is to truncate a non-trivial fraction of the output bits. Unfortunately, this method suffers from relatively poor exact security (and, of course, gives shorter output). The third solution is similar in spirit to the NMAC PRF solution above, and requires an independent compression function f'. which is then applied to $\textsc{Cascade}(M, f, IV)$. Unfortunately, it is not practically convenient to design two different compression functions f and f'.[8] Finally, the last method is similar to the HMAC PRF method above, and could be viewed as an ad hoc trick to avoid having two independent compression functions: it outputs $\textsc{Cascade}(\textsc{Cascade}(0^b M, f, IV), f, IV)$. Until our work, this was considered the method of choice, since only one compression function is used, the message can be processed on-line, and only two extra calls to f are made are made. With our on-line PFE, however, we end up making the same number of calls to f, but, arguably, obtain a result which is simpler to understand and explain (in terms of security).

Domain Extension of MACs. Another natural usage of the cascade mode is to extend the domain of a message authentication code (MAC) $f_k : \{0,1\}^{b+s} \to \{0,1\}^s$. This is different from the CBC-MAC and the cascade-PRF applications above because the building block here is only assumed to be unpredictable, as opposed to pseudorandom (indeed, it is easy to see that the CBC-MAC and the cascade constructions are *not* always secure if the block cipher or the b-bit PRF are only assumed unpredictable [AB99]). Still, we can still define Cascade-MAC$(k, M) = \textsc{Cascade}(M, f_k, 0^s)$, where our compression function f_k is keyed by the same key k throughout (so called dedicated-key setting [BR07]).

It was observed by Maurer and Sjödin [MS05] that the Cascade-MAC is secure if M is encoded in a prefix-free form. (The extension attack here is a bit more complicated; we refer to [BR07].) On the other hand, without using on-line PFE (which was conjectured to be too wasteful by Maurer and Sjödin), several techniques to overcome this problem are known. The simplest one is to use, once again, an additional key k', and apply $f_{k'}$ to the output of the Cascade-MAC(k, M) (see [MS05], which slightly optimizes the earlier suggestion of [AB99]). Without having two keys, several ad hoc solutions were developed by [MS05, BR07]. While offering comparable efficiency to using the basic cascade construction with our PFE, these solutions are arguably less intuitive than the basic cascade construction.

[6] Actually, one uses the so called *strengthened Merkle-Damgård Transform*, where the length of the message is appended as the last message block. However, this detail does not affect our discussion below, so we stick with "plain Merkle-Damgård", which is precisely the cascade mode.

[7] While this attack might appear theoretical, consider a simple PRF construction $F_k(M) = H(k, M)$. It is clearly secure if H is a random oracle, but completely insecure (due to the extension attack above) when H is implemented by the cascade mode.

[8] Ironically, one way to design two independent compression functions f_0 and f_1 from a single compression function f is to prepend 0 when calling f_0, and 1 — when calling f_1. However, this is *equivalent* to the classical "wasteful" PFE, where each but last block is prepended with 0, and the last block is prepended with 1. Clearly, our PFE will result in a much less wasteful solution.

We also notice that the dedicated key setting can be used for the domain extension of PRFs as well, when f_k is a assumed a PRF rather than a MAC (but one gets a PRF back as well). The discussion here is similar to the MAC case.

Multi-Property Preservation. As we have seen, in all the applications that we mentioned, applying the PFE followed by the cascade construction works. Thus, the PFE followed by the cascade construction can be seen as a *multi-property preserving* transform (MPPT). As advocated by Bellare and Ristenpart [BR06, BR07], devising MPPTs is advantageous in terms of reusing the existing implementations of hash functions, by designing compression functions which simultaneously satisfy several desired properties. However, since online PFEs were considered impractical, Bellare and Ristenpart [BR06, BR07] had to work relatively hard to design their MPPTs. Arguably, the "PFE-then-Cascade" is a much simpler MPPT to understand and explain than the MPPTs developed by [BR06, BR07]. Given that our work gives a very efficient PFE, our MPPT also offers roughly the same efficiency than the MPPTs of [BR06, BR07], so it could be an attractive alternative to those MPPTs.