Bachelor Thesis

Encoding Schemes for Peptide Mass Storage

by
Missaoui Yahya

Student ID: 7893687
s9946661@stud.uni-frankfurt.de
February 11, 2026

Submitted as part of the
Bachelor Thesis

First Examiner: Prof. Dr. Lena Wiese
Second Examiner: Prof. Dr. Ingo Ebersberger

at
Goethe University Frankfurt
Department of Computer Science and Mathematics

Abstract

Molecular storage is a promising option for long-term, energy-passive data archiving. While most prior work focuses on DNA, peptides offer a different substrate with a richer symbol space and mass-spectrometry readout constraints. This thesis asks whether DNA-oriented coding schemes can be transferred to peptide-based storage without changing their core logic, using only adaptations such as alphabet mapping, fixed peptide framing, and parameter tuning.

A modular evaluation pipeline is implemented that encodes files into fixed-length peptides over a restricted alphabet, applies a residue-level stochastic channel, and decodes using either a block-based Reed–Solomon (RS) outer code or a fountain-style peeling decoder. Results show that Huffman source coding, Yin–Yang-style constrained mapping, and LT fountain coding can be instantiated in the peptide setting. Under a uniform sequence-agnostic stress test, increased RS parity improves robustness and provides a conservative upper bound, whereas under a score-driven sequence-aware channel RS64 already yields stable recovery up to 1 MB. Constrained mapping improves PRR near the RS threshold (RS32) in the sequence-aware regime but reduces net rate due to its 2-bit payload mapping, and fountain coding is feasible in the tested error regime but exhibits steep runtime scaling with file size.

Contents

1 Introduction

1.1 Motivation and Problem Context

In recent years, there has been a rapid growth in the amount of digital data [1] generated across scientific, industrial and societal fields. Historically, hardware scaling trends enabled sustained improvements in hardware performance and density; however, as data generation continues to grow exponentially, these improvements are becoming less sufficient to resolve the increasing mismatch between data growth and hardware evolution. As a consequence, storing large amounts of digital data over a long period of time is becoming increasingly challenging. Traditional data storage technologies and methods are widely used and well established; however, they suffer from multiple limitations, particularly with respect to storage density, durability, and energy efficiency [2]. These aspects are fundamentally interdependent: pushing density tends to reduce physical noise margins, which requires additional redundancy and maintenance to preserve reliability, thereby increasing energy and operational overhead. Notably, the limited operational lifespan of these media leads to frequent data transfers and migration, resulting in additional costs, increased system complexity, and higher risks of data loss during transfer and maintenance operations.

In this context, the limitations of electronic and magnetic storage media have motivated research into alternative data storage technologies. Consequently research has increasingly focused on fundamentally different storage paradigms that aim to overcome the physical and operational constraints of conventional media.

1.2 Peptides as a Medium for Molecular Data Storage

Molecular data storage emerged as a promising approach to address long term data preservation challenges, while DNA based systems receiving the most attention to date.

Molecular storage media allow information to be stored without requiring a continuous energy supply, enabling data to remain preserved even in the absence of active hardware [3]. Among molecular storage approaches, DNA based systems have received the most attention, as DNA sequences reliably encode biological information and can remain stable for long pe-

riods under suitable storage conditions [4, 5]. In addition, DNA based storage benefits from decades of progress in synthesis and sequencing technologies, which provide a relatively mature and well understood technological basis [6, 7] . A further key advantage of molecular storage lies in its exceptionally high storage density, which can exceed that of electronic and magnetic storage media by several orders of magnitude [3, 5] .

DNA based storage have demonstrated that digital information can be encoded, stored, and retrieved using molecular substrates in a reliable way, even under experimental conditions that require the use of error correction and redundancy mechanisms. As a result, DNA based storage has become the reference point for molecular data storage research.

However, DNA is not the only biological polymer capable of representing digital information. Alternatively, peptides, which consist of sequences of amino acids, represent one such molecular storage medium [8]. In contrast to DNA, peptides can incorporate a substantially larger set of chemically distinct monomers because their synthesis and sequencing do not rely on enzyme recognition processes [8]. This larger effective symbol alphabet can increase flexibility in data encoding. From an information theoretic perspective, a larger alphabet can increase the amount of information encoded per symbol; however, the achievable gains depend critically on how encoding schemes interact with substrate specific noise and readout uncertainty. In addition, peptide synthesis and analysis are supported by established techniques originating from proteomics research, where high throughput workflows are already widely employed [8]. In particular, mass spectrometry based readout introduces measurement characteristics that differ fundamentally from DNA sequencing, as information is inferred from mass to charge distributions rather than discrete base identification.

In contrast to DNA storage, encoding strategies for peptide based systems remain comparatively underexplored, which limits the understanding of how existing molecular encoding principles perform when applied to peptide substrates [8]. Addressing these gaps is essential for understanding the potential and limitations of peptide based data storage.

1.3 Research Question and Scope

This thesis aims to examine whether encoding approaches developed for molecular data storage can be transferred to peptide sequences, in other words, whether the underlying design principles remain applicable when the molecular substrate and sequence alphabet change.

Can coding schemes originally developed for DNA based data storage, such as Huffman coding, DNA Fountain, and Yin-Yang coding, be applied to peptide sequences without requiring fundamental changes? In this thesis, “fundamental changes” refers to redesigning the core logic of a scheme; only minimal adaptations are considered, such as adjusting the

alphabet mapping, parameter settings, and constraint handling to match peptide sequence properties.

This work focuses on computational evaluation and does not include experimental wet-lab validation. The goal is to provide a structured assessment of the transferability and limitations of DNA based coding schemes when applied to peptide sequences, using implementation based experiments and quantitative metrics (e.g., encoding overhead and decoding reliability under modeled errors) to identify where direct transfer is feasible and where peptide specific modifications become necessary. To contextualize our research question, the following section summarizes existing work on molecular data storage and encoding schemes relevant to this thesis.

2 Related Work

2.1 Molecular Data Storage as an Alternative Paradigm

Molecular data storage approaches the same problem from a different direction [2] . Instead of storing bits in electronic states or magnetic domains, it represents information as sequences of chemical building blocks. Data is written through synthesis, and later recovered through analytical readout. The important point is what happens in between: once the molecules exist, keeping the information does not require continuous power or active device operation [2, 3]. Data longevity becomes primarily a property of chemical stability and storage conditions rather than a property of running hardware.

Prior work frequently highlights the high theoretical storage density and long-term stability of molecular substrates [3, 7]. Together, these properties make molecular systems attractive for archival scenarios in which durability is prioritized over access speed.

Given these motivations, research has focused on two linked questions: which molecular carriers are practical, and which encoding strategies allow reliable recovery in the presence of synthesis and readout imperfections. Biological polymers have drawn particular interest here. They offer regular, sequence-defined structures and can be analyzed with mature laboratory techniques developed in neighboring fields. This combination has made biological sequences a natural starting point for molecular storage research. This focus has shaped much of the existing research on molecular data storage.

2.2 DNA as a Medium for Molecular Data Storage

In much of the literature, storage using DNA sequences is implemented by distributing data across many short strands rather than relying on a single long molecule. Each strand typically contains a payload region and additional structure that supports reconstruction, such as an index or addressing information used to identify the strand within a pool [9]. This strand set perspective is central to system design, since reliable recovery depends not only on symbol-level accuracy but also on whether the relevant strands remain available after synthesis, storage, and sequencing.

Representative systems follow this pooled-oligonucleotide model while differing in how they enable reconstruction and selective retrieval. For example, large-scale demonstrations use explicit indexing and primer-based selection to support random access to individual files within a mixed pool [9]. Earlier work also explored architectures that support rewriting and targeted access by combining constrained addressing with editing operations [10]. These designs highlight that the DNA medium is typically accessed through operations on pools and subsets, not through sequential reads of a single continuous sequence.

From an encoding perspective, DNA storage must be designed for a noisy write–store–read pipeline, in which both within-strand corruption and strand-level availability effects can occur. The specific error sources and their implications for coding are discussed in Section 2.3.1; these assumptions motivate layered redundancy and the use of error-correction schemes tailored to molecular channels [2, 3, 11].

Taken together, these assumptions and constraints shape many coding approaches developed in the DNA storage literature. This is directly relevant for the present thesis: when transferring encoding schemes to peptide sequences, the alphabet, dominant constraints, and readout mechanism change, and it becomes necessary to identify which design elements remain applicable and which implicitly rely on DNA-specific conditions.

2.3 Encoding and Error Correction in Molecular Storage Systems

2.3.1 Error sources during writing and readout

Encoding strategies in molecular storage are strongly shaped by the fact that both writing and readout are imperfect. During chemical synthesis, errors can be introduced directly into the produced sequences. Reported synthesis issues include substitutions, insertions, and deletions, as well as reduced yield that can lead to missing sequences in the final pool [12, 13]. Readout introduces additional distortion. For storage using DNA sequences, the dominant error types depend on the sequencing technology, but commonly discussed effects include substitutions, insertions and deletions, and uneven coverage, where some strands are undersampled or not observed at all [4, 10]. The resulting channel therefore combines two levels of uncertainty: corruption within a strand and the loss of entire strands from the pool [2].

Because these effects appear repeatedly across experimental demonstrations, prior work typically relies on layered protection rather than a single mechanism. Compression can reduce the number of molecules that must be synthesized, but it also increases sensitivity to corruption if the compressed bitstream loses synchronization. Error correction is therefore

usually applied across many strands, and it is often combined with design rules that avoid sequence patterns known to interact poorly with synthesis and sequencing.

2.3.2 Redundancy across strands: Reed–Solomon and interleaving

A common approach in the literature is to add redundancy across a block of strands using classical block codes. Reed–Solomon codes are frequently used as an outer code because they operate on symbols over finite fields and can correct a bounded number of symbol errors or erasures within a block [14, 9]. This is well matched to pooled DNA storage, where strand dropout behaves like an erasure and where remaining strands may still contain within-strand corruption.

Interleaving can be used together with outer coding to spread localized failures across multiple codewords. If certain strands are more errorprone due to synthesis variability or sampling effects, interleaving reduces the chance that a single block is dominated by those failures. Work on indel-resilient decoding further emphasizes that insertions and deletions can be particularly damaging because they disrupt symbol alignment, which motivates careful layering of redundancy and reconstruction procedures [11].

2.3.3 Handling strand loss: fountain-style schemes

Strand loss and uneven sampling are central concerns in large pools, and they motivate erasure-tolerant constructions. Fountain codes produce a stream of encoded packets such that the original data can be recovered once a sufficient number of packets is collected. DNA Fountain adapts this idea to storage using DNA sequences by combining a fountain construction with indexing and screening rules that reject sequences predicted to be difficult to synthesize or sequence [15]. In this view, dropout is treated as a first-class design constraint: instead of assuming all strands will be present, the coding strategy is designed to remain decodable when only a subset is recovered.

2.3.4 Constraint-aware mapping: the Yin–Yang codec

In addition to redundancy, prior work often uses transcoding rules that shape the generated sequences themselves. Many approaches aim to avoid patterns associated with higher error rates, such as long homopolymers or extreme GC content (the fraction of guanine and cytosine bases in a strand). The Yin–Yang codec is a representative example that encodes bits into nucleotides while enforcing compatibility with practical synthesis and sequencing constraints through paired encoding rules [16]. This illustrates a broader design principle: reduc-

ing the effective noise of the physical channel can be achieved not only by adding redundancy, but also by generating sequences that are less likely to trigger synthesis and readout problems.

The encoding approaches reviewed above were largely developed around nucleotide alphabets and sequencing pipelines. When transferring these ideas to peptide sequences, both the alphabet and the readout mechanism change, and synthesis can introduce its own characteristic by-products such as truncations and deletions [17]. The following section therefore reviews prior work on storage using peptide sequences and discusses which error and constraint assumptions differ most from the DNA setting.

2.4 Storage using peptide sequences in the literature

Work on storage using peptide sequences is still much smaller than the DNA literature, but it establishes a concrete end-to-end workflow: map digital symbols to amino acids, synthesize the resulting peptides, and recover the data by sequencing the peptides with liquid chromatography coupled to tandem mass spectrometry (LC–MS/MS) [8]. In this setting, the central engineering problem is not only how to encode bits efficiently, but how to choose sequences that remain synthesizable and readable with high fidelity.

Ng et al. demonstrated the feasibility of peptide-based digital storage and provided a reference design for handling mixed peptide pools [8]. They split the payload into short peptides and add an address component so that peptides can be reordered during decoding [8]. A key practical point in their study is that data-bearing peptides require near-complete sequence recovery, unlike typical proteomics workflows where peptides can often be identified even at low coverage by searching against sequence databases [8]. To support reliable recovery, Ng et al. combined careful sequence design with explicit error protection (reporting error correction that tolerates missing or incorrect residues) and developed custom software tailored to their peptide design [8].

Peptide synthesis introduces its own failure modes, and these are sequence dependent. Solid phase peptide synthesis involves many repeated coupling and deprotection steps, and some sequences accumulate side products or incomplete sequences more readily than others, which reduces the fraction of correct full-length product [17]. In the data storage demonstration by Ng et al., this reality already influenced design choices: they restricted the amino acid set and avoided or limited residues that made synthesis or detection less reliable for their protocol [8]. More recently, Gutman et al. proposed PepSySco, a model that predicts the likelihood that a given sequence can be successfully produced with Fmoc solid phase peptide synthesis [18]. For peptide storage, this type of predictor can play a similar role to sequence

screening in DNA storage: it provides a practical constraint signal that can be used to reject candidates likely to yield poor synthesis outcomes.

Readout by tandem mass spectrometry does not observe a sequence directly. Instead, the sequence must be inferred from fragmentation patterns, and accuracy depends on spectrum quality, fragmentation completeness, and the algorithm used for reconstruction [19]. Certain ambiguities are also fundamental. A well-known example is that leucine and isoleucine are isobaric and cannot be distinguished by mass alone in standard workflows, which motivates either additional experimental strategies or alphabet choices that avoid such ambiguity when high-fidelity decoding is required [20, 21]. These constraints push peptide storage designs toward carefully chosen symbol alphabets and decoding pipelines rather than relying on generic proteomics tooling.

Beyond direct digital storage demonstrations, peptide sequences are also used as information carriers in peptide encoded library systems, where decoding relies on MS/MS and where the choice of a non-isobaric encoding alphabet and explicit tag design rules are emphasized to improve sequence recovery [22]. Recent work has also begun to address practical aspects such as preservation of peptide material for long-term retention and alternative mapping strategies that increase effective coding density under synthesis constraints [23, 24]. Taken together, the peptide literature suggests three recurring design pressures: sequence-dependent synthesis reliability, readout ambiguity and algorithmic uncertainty in MS/MS reconstruction, and the need for explicit addressing and error protection when data are distributed across many short peptides [8, 18, 19].

2.5 Summary of Prior Work and Research Gap

Molecular data storage is commonly framed as an archival alternative to conventional electronic media, motivated by the high theoretical density and long-term stability of molecular substrates [2, 3, 7]. Most DNA-based systems operationalize this idea by spreading information across large pools of short strands. These strands typically carry an explicit address or index, which supports reconstruction from a mixed pool and enables selective access to subsets of the data [9, 10]. At the same time, the write–store–read pipeline combines two distinct uncertainties: symbol errors within strands and strand-level effects such as uneven sampling or complete dropout. As a result, DNA storage designs rarely rely on a single protective mechanism; instead, they layer redundancy across strands, erasure tolerance, and constraint-aware mapping to mitigate synthesis and sequencing failures [14, 15, 16, 11].

The peptide literature is smaller, but it points to a different set of practical constraints. On the writing side, synthesis feasibility is strongly sequence dependent [17, 18]. On the

reading side, LC–MS/MS does not directly output a sequence; reconstruction is inferred from fragmentation spectra and inherits intrinsic ambiguities, including isobaric residues [19, 20, 21]. Recent contributions extend this line of work toward more practical deployment, for example by improving preservation and by proposing mapping strategies that increase effective coding density under synthesis constraints [23, 24].

Even with these advances, it is still unclear how far DNA-oriented encoding schemes can be transferred to peptide-based pipelines. Many constructions in the DNA storage literature are designed and evaluated under assumptions tied to the nucleotide alphabet, typical biochemical constraints, and sequencing-specific error patterns [2, 4]. Switching to peptides changes both the symbol set and the readout mechanism, which reshapes the dominant error models and constraints during decoding [8, 19]. In practice, reliability is achieved through a stack of components rather than an isolated code: source coding (compression), framing and addressing, constrained mapping, and outer error correction. For peptide storage, the coupling between these layers becomes especially important because synchronization loss and MS/MS reconstruction ambiguity can affect large portions of the decoded bitstream, and because feasibility depends tightly on the chosen alphabet and mapping strategy.

This thesis addresses the research gap by systematically evaluating encoding-scheme components under a peptide-oriented pipeline. The aim is to determine which design elements from DNA storage remain effective when translated to peptide sequences, which rely on DNA-specific conditions, and where adaptation is required to accommodate peptide synthesis constraints and LC–MS/MS readout characteristics.

3 Methodology

3.1 Study Design, Scope, and Notation

This thesis investigates whether encoding strategies developed for DNA-based molecular data storage can be transferred to peptide-based storage. To enable a controlled comparison, a software evaluation pipeline is implemented that (i) encodes digital files into fixed-length peptide sequences over a restricted amino-acid alphabet, (ii) applies an amino-acid-level channel model, and (iii) decodes the perturbed sequences to assess end-to-end reconstruction reliability. The pipeline is modular, enabling different encoding schemes to be evaluated within the same framework.

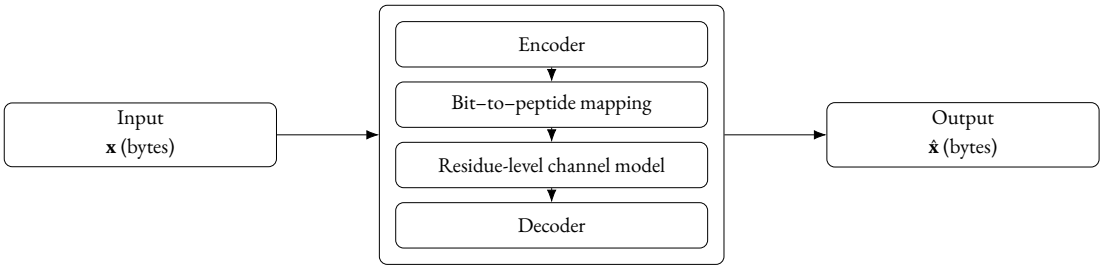


Figure 3.1: Overview of the evaluation pipeline.

3.1.1 Experimental design and assumptions

The objective is to evaluate, in a peptide setting, encoding schemes that are widely used in DNA storage and to quantify their robustness under residue-level perturbations. Three scheme families are considered: a Huffman-based baseline, a Fountain-code-based scheme, and a Yin–Yang-style constrained scheme. Each scheme produces a peptide sequence representation of the input; decoding performance is then quantified using the metrics defined in this chapter. The evaluation is conducted in software and focuses on coding and decoding behavior under a defined stochastic channel model.

Varied factors.

- **Encoding scheme:** Multiple encoding schemes are compared under a shared pipeline backbone.
- **Outer redundancy configuration:** Redundancy is varied as an experimental factor, but its realization is scheme-dependent. For RS-protected configurations (e.g., Yin–Yang with an outer Reed–Solomon layer), redundancy is parameterized by the number of parity peptides r added per block (e.g., $r \in \{8, 16, 32, 64, 128\}$). For fountain-based configurations, redundancy is parameterized by an overhead factor ρ, i.e., transmitting $(1 + \rho)$ encoded droplets relative to the k_{F} source symbols. For each scheme, the corresponding redundancy parameter (either r or ρ) is swept while all remaining pipeline components and channel parameters are kept fixed, to quantify the redundancy–reliability trade-off under identical channel conditions.
- **Input size:** Files spanning a wide size range (from bytes up to megabytes) are used to expose size-dependent failure behavior.

Controlled factors. The following factors are kept fixed across all experiments unless stated otherwise:

- **Peptide alphabet and mapping interface:** A restricted peptide alphabet Σ with $|\Sigma| = 8$ is used throughout, corresponding to $\log_2 |\Sigma| = 3$ bits of symbol capacity per residue.
- **Peptide length and framing:** Peptides have a fixed length L (default $L = 18$). An optional index prefix may be enabled, which reduces payload capacity accordingly.
- **Evaluation criteria:** The same end-to-end success criterion and the same evaluation metrics are applied to all schemes.

Channel settings (evaluation scenarios). Two fixed channel parameterizations are evaluated:

- **Uniform (sequence-agnostic) channel:** a stress-test setting with sequence-independent error behavior.
- **Score-driven (sequence-aware) channel:** a setting in which per-peptide error rates depend on predicted sequence quality (Section 3.5).

All cross-scheme comparisons are interpreted *within* the same channel setting. The uniform setting is used to establish a conservative reference point, while the score-driven setting is used for the main comparisons.

Success criterion. A run is considered successful if the reconstructed byte sequence matches the original input exactly. Let $\mathbf{x}$ denote the original byte sequence and $\hat{\mathbf{x}}$ the decoded output. Success is defined as:

$$\text{success} = \mathbb{1}[\hat{\mathbf{x}} = \mathbf{x}].$$

No additional checksum layer is used in the RS-protected pipelines; therefore correctness is evaluated solely by end-to-end equality of bytes. The Fountain configuration includes a per-droplet cyclic redundancy check (CRC) that is used only to discard corrupted droplets (treating them as erasures); end-to-end correctness is still evaluated by byte equality.

Evaluation assumptions. The evaluation makes simplifying assumptions that follow from the implemented pipeline:

- **Metadata is available to the decoder.** Parameters required for decoding (e.g., block structure, padding information, and optional indexing settings) are assumed to be known during decoding. The current pipeline does not embed this metadata into peptide payloads.
- **One observation per peptide.** Each peptide is evaluated as a single observed sequence after corruption. The pipeline does not simulate multiple reads, confidence scores, or consensus calling.
- **Stochastic channel without fixed seeding by default.** Error application is randomized; no fixed seeding strategy is used unless explicitly enabled for debugging or replication.

3.1.2 Notation

$\mathbf{x}$, $\hat{\mathbf{x}}$ Original input bytes and decoded output bytes.

Σ Restricted peptide alphabet, $|\Sigma| = 8$, $\Sigma = \{\texttt{A},\texttt{V},\texttt{L},\texttt{S},\texttt{T},\texttt{F},\texttt{Y},\texttt{E}\}$ (chosen as in [8]).

L Peptide length in residues.

k_{RS}, r, n RS block parameters: k_{RS} data peptides, r parity peptides, $n = k_{\text{RS}} + r$.

B Number of bytes used to represent one peptide in the RS layer after packing and byte alignment.

$p_{\text{loss}}, p_{\text{mut}}, p_{\text{ins}}, p_{\text{shuf}}$ Per-residue probabilities for deletion, substitution, insertion, and local shuffling.

L_{idx} Optional index-prefix length (residues) when indexing is enabled.

L_{payload} Payload residues per peptide, typically $L_{\text{payload}} = L - L_{\text{idx}}$.

ϕ, ϕ^{-1} Fixed 3-bit ↔ residue mapping over Σ (baseline mapping / RS serialization) and its inverse.

ψ, ψ^{-1} Yin–Yang constrained payload mapping and its inverse.

ρ Fountain overhead factor: transmit $(1 + \rho)$ droplets per k_{F} source symbols.

k_{F} Number of fountain source symbols (packets).

d_{int} Optional interleaving depth applied before RS block formation.

$P, \tilde{P}$ Transmitted peptide list P and observed (corrupted) list $\tilde{P}$ under the channel.

N Number of peptides in a peptide list, $P = (P^{(1)}, \ldots, P^{(N)})$.

$\pi^{(j)}$ The j-th peptide, written as $\pi^{(j)} = (p_1^{(j)}, \ldots, p_L^{(j)})$ with residues $p_i \in \Sigma$.

u Uniform random draw $u \sim \text{Unif}(0, 1)$ used by per-position Bernoulli operators.

$Q(\pi)$ (Score-driven channel) Predicted peptide quality/score used to derive per-peptide error rates.

$p_{\text{min}}, p_{\text{max}}$ (Score-driven channel) Lower/upper bounds for induced base error probabilities.

RSr Shorthand for a Reed–Solomon configuration with r parity symbols (e.g., `RS8`, `RS16`, …).

3.2 Input Data and Experimental Setup

3.2.1 Input corpus

The evaluation uses a corpus of files whose sizes follow a power-of-two schedule from 1 byte up to 1 MB. To keep runtime and memory bounded, experiments are limited to inputs of at most 1 MB. Larger files can be handled by partitioning the byte stream into fixed-size chunks (e.g., 1 MB per chunk) and treating each chunk as an independent encoding/decoding instance; the original file is then reconstructed by concatenating decoded chunks in their recorded order.

Input files are generated synthetically and stored on disk prior to evaluation. For each size 2^e with $e \in \{0, \ldots, 20\}$, a file of exactly 2^e bytes is created by sampling from the operating system entropy source and filtering the stream to a restricted set of printable ASCII characters (letters, digits, whitespace, and common punctuation). The resulting stream is truncated to the target size. Because filtering is applied, the byte distribution is not uniform over $\{0, \ldots, 255\}$ but biased toward the allowed character set.

The same persisted corpus is reused across all configurations and runs. Reproducibility is ensured by keeping the generated input files fixed throughout the experiment campaign (rather than regenerating them for each run).

3.2.2 Experimental Setup and Run Definition

A primary comparison dimension is the redundancy setting, using the scheme-specific parameterization introduced in Section 3.1.1 (RS parity size r for RS-based configurations and overhead factor ρ for fountain-based configurations). Selected configurations additionally enable interleaving with fixed depth to spread correlated corruption across blocks, and optional indexing to support peptide placement during decoding. All other pipeline components follow the controlled factors in Section 3.1.1.

Experiments use the residue-level channel defined in Section 3.5 under two fixed parameterizations: a uniform (sequence-agnostic) setting and a score-driven (sequence-aware) setting. The operator definitions, their application order, and the per-residue probabilities are specified there. Cross-scheme comparisons are evaluated within the same channel setting.

A run is defined by the tuple *(input file, redundancy configuration, channel parameters)*. The pipeline in Fig. 3.1 is executed once per run (encode → channel → decode), and correctness is evaluated using the success criterion in Section 3.1.1. For each run, the implementation logs one row in a tabular results file, including the input identifier, encoding scheme, redundancy setting (RS parity profile r or fountain overhead ρ), peptide length and indexing parameters, the channel probabilities (p_{loss}, p_{mut}, p_{ins}, p_{shuf}), original and decoded sizes, and the error metrics defined later in this chapter.

The channel model is stochastic and is driven by a pseudo-random number generator that is currently initialized without an explicit seed. Consequently, repeated executions of the same file–configuration pair may yield different corruption realizations and outcomes.

Uniform stress tests use multiple independent channel draws per setting, whereas score-driven runs use a single draw per file–configuration pair for coverage; seeds are not logged, so exact replay is not enabled in the current script.

3.3 Encoding Pipeline

An input byte sequence $\mathbf{x}$ is transformed into a peptide stream over the restricted alphabet Σ with target length L and may be protected by an outer redundancy mechanism (e.g., an RS parity layer or fountain-style overhead, depending on the configuration). The notation follows Section 3.1.2.

3.3.1 Source coding (Huffman baseline)

For the Huffman-baseline configuration, the input file is treated as an arbitrary byte sequence $\mathbf{x} \in \{0, \ldots, 255\}^m$, where $m = |\mathbf{x}|$. A per-file Huffman code is constructed from the empirical byte distribution of that file, and the file is then compressed using this code, yielding a variable-length compressed representation.

Implementation-wise, Huffman coding is realized using the `dahuffman`[1] library. The encoder constructs a codec , i.e., the Huffman alphabet consists of byte symbols and the codebook is derived from the file itself. The file is then encoded , which produces a byte-aligned compressed bytestream. For subsequent stages that operate on bits, this bytestream is converted into a binary string representation (e.g., `'0101...'`) using a deterministic bytes-to-bitstring conversion.

Implementation note: The Huffman baseline uses `dahuffman` to construct a per-file Huffman codec directly from the input bytes. Internally, the library derives the symbol distribution by a single pass over the provided data and counting occurrences of each byte value using `collections.Counter`. Since iterating over a Python `bytes` object yields integers in $\{0, \ldots, 255\}$, the Huffman alphabet is exactly the set of byte symbols observed in the file and their empirical frequencies. Based on these counts, `dahuffman` builds a standard prefix-free Huffman code (via repeated merging of the two least frequent nodes using a priority queue) and stores a code table mapping each byte symbol to a variable-length bit codeword.

Although the code is bit-oriented, the library outputs a `bytes` bytestream. To make decoding unambiguous when the final codeword does not align to an 8-bit boundary, the implementation uses an explicit end-of-stream marker in the code tree and emits it at the end of encoding. During decoding, the bit reader stops once this marker is encountered, so any remaining padding bits in the last output byte are ignored safely.

Consistent with the study scope (Section 3.1.1), the decoder is assumed to have access to the same Huffman codebook/codec object used at encoding time; codebook transmission

[1] https://pypi.org/project/dahuffman/

and its overhead are not modeled. Decoding reverses the above steps: the stored bitstring is converted back to bytes and decoded to recover **x**.

No additional whitening or scrambling is applied. Consequently, any statistical structure induced by Huffman coding is preserved and propagated into the subsequent bit-to-residue mapping stage.

3.3.2 Yin–Yang-style constrained mapping

The Yin–Yang codec uses a redundant 2-bit-to-residue payload mapping that introduces controlled encoding freedom while remaining exactly decodable. The payload bitstream is partitioned into consecutive 2-bit symbols $d_k \in \{0,1\}^2$. Each symbol determines two candidate residues via the fixed "Yang" mapping ψ (Table 3.2), and the encoder selects one candidate using the "Yin" rule below.

Table 3.2 lists the candidate pairs $\psi(d_k)$. Decoding uses a deterministic inverse mapping $\psi^{-1} : \Sigma \to \{0,1\}^2$ where both residues in each pair map back to the same 2-bit value; therefore, encoding has two choices per symbol while decoding remains unambiguous.

For the next symbol d_k, let $\{c_0, c_1\} = \psi(d_k)$ be the two candidates. The encoder maintains the current payload prefix (within the current peptide) and chooses the candidate with smaller penalty:

$$c^\star = \arg\min_{c \in \{c_0, c_1\}} \Pi(c \mid \text{current payload prefix}).$$

The penalty $\Pi(\cdot)$ is an offline heuristic that approximates "synthesis-friendly" behavior (e.g., high PepSySco-like quality) without querying an external scoring service during encoding.

Hard constraints are enforced by adding a very large penalty (effectively forbidding the choice) whenever a candidate would violate:

- **No long identical runs:** forbid more than 2 identical residues in a row.
- **No long hydrophobic runs:** forbid more than 2 consecutive residues from $H = \{\mathtt{V}, \mathtt{L}, \mathtt{F}, \mathtt{Y}\}$.
- **No long `E` runs:** forbid more than 2 consecutive `E`.
- **Aromatic-count constraint:** let $A = \{\mathtt{F}, \mathtt{Y}\}$ denote the set of aromatic residues. The per-peptide aromatic count is constrained by

$$N_A(\pi) \le \kappa_A(L_{\text{payload}}), \qquad \kappa_A(L_{\text{payload}}) = \max\big(1, \min(3, \lfloor L_{\text{payload}}/6 \rfloor)\big),$$

  where $N_A(\pi) = \sum_{i=1}^{|\pi|} \mathbf{1}[p_i \in A]$.

- **E-count constraint:** the number of E residues per peptide is constrained by

$$N_E(\pi) \;\le\; \kappa_E(L_{\text{payload}}), \qquad \kappa_E(L_{\text{payload}}) = \max\!\big(2, \min(6, \lfloor L_{\text{payload}}/3 \rfloor)\big),$$

  where $N_E(\pi) = \sum_{i=1}^{|\pi|} \mathbf{1}[p_i = \texttt{E}]$.

Soft preferences contribute small additive penalties/bonuses, including (i) penalizing hydrophobic residues in H, (ii) penalizing aromatics in A, (iii) preferring S and T, and (iv) a small preference against immediate repeats. These rules bias the emitted peptides away from long hydrophobic stretches and excessive aromatics while preserving exact decodability via ψ^{-1}.

3.3.3 Bitstream to peptides (packing, padding, indexing, RS interface)

Downstream stages operate on (i) residue-symbol streams and (ii) a fixed-size byte-vector interface for RS processing. Therefore, padding/normalization may be introduced at two boundaries:

- **Payload-symbol alignment.** For the Huffman baseline (fixed 3-bit mapping), the bitstream is padded at the end with 0-bits to a multiple of 3 so it can be partitioned into consecutive 3-bit symbols; the number of added bits is recorded and removed after payload reconstruction during decoding. For the Yin–Yang codec (2-bit payload symbols), the bitstream is padded with trailing 0-bits to a multiple of 2; in the current pipeline (bytes → bits), this padding is typically zero but is retained for completeness and removed during decoding.

- **RS symbol representation.** For RS-based configurations, each peptide is converted into a fixed-size byte-vector representation prior to RS encoding/decoding. This conversion may require deterministic normalization/padding as needed to meet the fixed byte-vector length; the necessary information for consistent inverse conversion is retained for decoding.

Huffman baseline (fixed 3-bit mapping). After bitstream padding and alignment, the payload is partitioned into consecutive 3-bit groups. Each 3-bit group is interpreted as an integer in $\{0, \ldots, 7\}$ and mapped to a residue in Σ via a fixed bijection $\phi : \{0,1\}^3 \to \Sigma$. The evaluation uses the fixed mapping shown in Table 3.1, i.e., $\phi(000) = \texttt{A}$, $\phi(001) = \texttt{V}, \ldots, \phi(111) = \texttt{E}$.

Yin–Yang payload mapping. For the Yin–Yang codec, payload mapping follows Section 3.3.2. Each payload residue carries exactly 2 bits (before indexing and outer-code overhead).

In both cases, the resulting payload residue stream is chunked sequentially into peptides of target payload length $L_{\text{payload}} = L - L_{\text{idx}}$ when indexing is enabled (Section 3.3.3), or

$L_{\text{payload}} = L$ otherwise. The final data peptide may contain fewer than L_{payload} residues if the residue stream does not end on a peptide boundary. No residue-level padding is added at this stage; instead, per-peptide length information is retained so that reconstructed data peptides can be trimmed consistently after outer-code decoding.

Table 3.1: Restricted amino-acid alphabet Σ (size 8) and fixed 3-bit mapping ϕ used for bit-to-residue conversion (baseline mapping and RS serialization).

3-bit code	Residue	3-bit code	Residue
000	A	100	T
001	V	101	F
010	L	110	Y
011	S	111	E

Table 3.2: Yin–Yang "Yang" mapping ψ: each 2-bit payload symbol maps to a pair of candidate residues. The encoder selects one residue from the pair (Yin rule), while the decoder maps either residue back to the same 2-bit value.

2-bit symbol	Candidate pair	Decoder inverse
00	{F, E}	F, E $\mapsto$ 00
01	{Y, S}	Y, S $\mapsto$ 01
10	{V, T}	V, T $\mapsto$ 10
11	{L, A}	L, A $\mapsto$ 11

In some configurations, each data peptide starts with an index prefix of length L_{idx} residues that encodes the peptide position within the encoded sequence. This reduces the payload capacity per data peptide from $3L$ to $3(L - L_{\text{idx}})$ bits for the baseline mapping. For Yin–Yang, the payload capacity is $2(L - L_{\text{idx}})$ bits, while the index prefix remains encoded using the fixed 3-bit mapping ϕ and is treated as metadata (removed before Yin–Yang payload reconstruction). Parity peptides produced by the RS layer do not require an interpretable index. When indexing is enabled, the decoder uses index information only to place data peptides (and to identify missing positions as erasures); the prefix of parity peptides is ignored by the placement procedure.

RS operates on byte symbols. For peptide-level RS protection, each peptide is mapped to a fixed-length byte vector used as the RS symbol representation. This serialization uses the fixed 3-bit mapping $\phi^{-1} : \Sigma \to \{0,1\}^3$ (Table 3.1) *regardless of the payload codec* (Huffman baseline, Yin–Yang, etc.). Internally, peptides are first normalized to the configured peptide

length L and then packed into bytes (3 bits per residue), with padding to the next byte boundary. For peptide length L, the resulting number of bytes per peptide is

$$B = \left\lceil \frac{3L}{8} \right\rceil.$$

For the default $L = 18$, this yields $B = \lceil 54/8 \rceil = 7$ bytes per peptide. The encoder retains the information required to invert the normalization after RS decoding so that reconstructed data peptides can be trimmed consistently before peptide-to-bit reconstruction. Peptides are marked as erasures during RS decoding if they are missing/empty, exceed length L, contain residues outside Σ, or (when indexing is enabled) carry an invalid index prefix that prevents consistent placement.

After the RS layer outputs corrected data peptides, index prefixes are removed if present and the payload residues are concatenated. Each residue is mapped back to 2-bit symbols using ψ^{-1}, yielding the reconstructed payload bitstream. The recorded payload-padding bits (typically zero) are removed, and the bitstream is converted to bytes and truncated to the original file length to recover $\mathbf{x}$.

3.4 Outer Redundancy and Error Protection

3.4.1 Reed–Solomon outer code

An outer Reed–Solomon (RS) layer can be applied to the peptide stream produced by an encoding scheme. RS operates on the byte-level representation of peptides introduced in Section 3.3.3 and is used to correct missing or corrupted peptides at the block level.

RS protection is applied independently to consecutive blocks of peptides. Each block contains k_{RS} data peptides and r parity peptides, yielding total block length

$$n = k_{\mathrm{RS}} + r.$$

The parity parameter r controls redundancy and is evaluated over multiple profiles, while k_{RS} is held constant across profiles (default $k_{\mathrm{RS}} = 32$ unless stated otherwise). RS coding is performed over $\mathrm{GF}(2^8)$, so the maximum code length is $n \leq 255$; when $n < 255$, a shortened RS configuration is used.

For RS processing, each peptide is represented as a fixed-length byte vector of length B (Section 3.3.3). This representation is internal to error correction; the storage abstraction remains a sequence of peptides over the residue alphabet Σ.

RS redundancy is applied column-wise across the B byte positions. For one block, let the k_{RS} data peptides be represented as byte vectors and arranged into a matrix

$$\mathbf{M} \in \{0, \ldots, 255\}^{k_{\mathrm{RS}} \times B},$$

where each row corresponds to a peptide and each column to a byte position within the packed representation.

For each column $j \in \{1, \ldots, B\}$, an RS encoder over GF(2^8) maps the k_{RS} data bytes $(M_{1,j}, \ldots, M_{k_{\mathrm{RS}},j})$ to r parity bytes $(M_{k_{\mathrm{RS}}+1,j}, \ldots, M_{k_{\mathrm{RS}}+r,j})$. Collecting parity bytes across all B columns yields r parity byte vectors of length B, which are mapped back to r parity peptides using the same packing and residue mapping conventions as for data peptides.

Because a single peptide corresponds to one row of $\mathbf{M}$, peptide-level corruption can manifest as correlated byte errors across multiple RS codewords (one per column), i.e., as a burst affecting several RS symbols in parallel.

The number of data peptides produced by the mapping stage is not necessarily a multiple of k_{RS}. If the last block contains fewer than k_{RS} data peptides, the encoder applies the same RS profile with an effective data length $k_{\mathrm{last}} < k_{\mathrm{RS}}$ and records k_{last} as metadata for decoding. No explicit end-of-stream marker is appended at the peptide level; termination is determined by the recorded block structure together with the padding metadata from Section 3.3.3.

Selected configurations enable interleaving with depth d_{int} (e.g., $d_{\mathrm{int}} = 4$). In the implementation, interleaving permutes the peptide sequence (and associated per-peptide metadata such as stored lengths, where applicable) *before* RS block formation and RS encoding. RS parity is therefore computed over the interleaved ordering.

3.4.2 Systematic LT fountain coding

As an alternative to RS-based protection, the pipeline supports a fountain-based configuration that provides rateless redundancy at the packet level. The input byte sequence is partitioned into k_{F} fixed-length source symbols (with zero-padding in the final symbol if required). The encoder then generates a stream of fixed-length droplets. Each droplet corresponds to one XOR equation over a subset of the k_{F} source symbols and carries an integrity check (CRC) so that corrupted droplets can be discarded at the decoder.

Redundancy is parameterized by an overhead factor ρ, meaning that the encoder transmits more droplets than source symbols. In addition, a minimum droplet budget is enforced for very small inputs so that decoding is not underdetermined even when ρ is small. Fountain evaluation follows the corpus input-size limit in Section 3.2.1.

Droplets are represented as fixed-length byte packets whose bit length is chosen to align with the peptide payload bit length under the fixed 3-bit residue packing used by the RS serialization and the fountain configuration. Let L be the peptide length and L_{idx} the optional index-prefix length, so the payload residue length is $L_{\text{payload}} = L - L_{\text{idx}}$ and the payload capacity per peptide is

$$C_{\text{pep}} = 3L_{\text{payload}} \quad \text{bits.}$$

The droplet packet size in bytes is chosen as the smallest positive integer b_{drop} such that $8b_{\text{drop}}$ is divisible by C_{pep}, ensuring that each droplet spans an integer number of peptides.

Each droplet packet consists of: (i) a compact header containing a seed and a degree, (ii) a body containing the bytewise XOR of the selected source symbols, (iii) zero padding inside the protected body if needed to keep packet length fixed, and (iv) a CRC computed over the preceding bytes. The subset of source-symbol indices is not transmitted explicitly; instead, it is reconstructed deterministically from the header. For degree 1, the index is derived directly from the seed (e.g., by modular reduction). For degrees larger than 1, a pseudorandom generator initialized from the seed selects the requested number of distinct indices from $\{0, \ldots, k_{\text{F}} - 1\}$. The construction is systematic: it first emits k_{F} degree-1 droplets that directly carry each source symbol, and then emits additional random droplets until the target droplet count implied by ρ is reached. For the random part, degrees are sampled from a robust soliton distribution over $\{1, \ldots, k_{\text{F}}\}$. Droplet packets are concatenated into a byte stream and converted into peptides using the existing fixed 3-bit packing and residue mapping (Section 3.3.3 and Table 3.1), with optional index prefixes handled as described in Section 3.3.3. Fountain decoding is described in Section 3.6.3.

3.5 Residue-Level Channel and Error Model

This section defines the stochastic channel that perturbs peptide sequences after encoding (and, where enabled, interleaving and outer protection). The channel operates at the residue level and is applied independently to each peptide; it does not change the global peptide order.

3.5.1 Channel definition, operators, and parameterization

Let $\pi = (p_1, \ldots, p_\ell)$ be an encoded peptide of length $\ell \leq L$. The channel produces an observed sequence $\tilde{\pi}$ by applying four residue-level operators. The resulting $\tilde{\pi}$ may have length different from both the target length L and the original length ℓ.

Error operators.

- **Residue deletion (Alg. 1).** For each residue position i, the residue p_i is removed with probability p_{loss}. This reduces sequence length and can yield very short (or empty, $|\tilde{\pi}| = 0$) observed peptides for large deletion rates.

- **Residue substitution (Alg. 2).** With probability p_{mut}, a residue p_i is replaced by a different symbol drawn from Σ. Substitution preserves length and maintains alphabet validity under the alphabet-restricted model.

- **Residue insertion (Alg. 3).** With probability p_{ins} per position, an additional symbol from Σ is inserted locally (before or after an existing residue, according to the implementation). Insertions increase sequence length and can create $|\tilde{\pi}| > L$.

- **Local shuffle (adjacent swap; Alg. 4).** With probability p_{shuf} per eligible position, two neighboring residues are swapped. This preserves length and symbol multiset but changes local order. Shuffling is confined within a peptide and does not permute peptide order across the encoded sequence.

Operators are applied in the fixed order

$$\text{deletion} \rightarrow \text{substitution} \rightarrow \text{insertion} \rightarrow \text{local shuffle}.$$

Within each operator, trials are modeled as independent Bernoulli events at the residue-position level. The model abstracts away correlations such as position-specific biases or context-dependent error rates.

Two parameterizations of $(p_{\text{loss}}, p_{\text{mut}}, p_{\text{ins}}, p_{\text{shuf}})$ are provided.

(1) Score-driven, sequence-aware parameterization (PepSySco-based). For each peptide sequence π, a peptide synthesis score $Q(\pi) \in [0,1]$ is obtained from PepSySco [2], where larger values indicate a higher predicted likelihood of successful synthesis [18]. The score is converted into a per-peptide base error rate

$$p(\pi) = p_{\min} + (1 - Q(\pi)) \cdot p_{\max}, \qquad p_{\max} = 0.02,\ p_{\min} \approx 0.00$$

and the per-residue operator probabilities used for that peptide are set to

$$p_{\text{loss}}(\pi) = p(\pi), \qquad p_{\text{mut}}(\pi) = p_{\text{ins}}(\pi) = p_{\text{shuf}}(\pi) = \frac{p(\pi)}{2}.$$

[2] https://tools.iedb.org/pepsysco/

Thus, peptides with lower synthesis scores are exposed to higher corruption rates, while high-scoring peptides experience fewer errors. This sequence-aware channel enables evaluation of mapping strategies whose goal is to generate "easier" peptide sequences, because the channel response depends on the produced sequence content.

Implementation detail. PepSySco queries are executed in batch mode by submitting all peptides in one request (one sequence per line). The returned response is parsed to map each peptide to its score $Q(\pi)$ and is saved for record-keeping. The implementation prints, per peptide, the original sequence, its score, the derived probabilities, and the post-channel sequence for traceability.

(2) Uniform, sequence-agnostic fallback parameterization. If score-based parameterization is unavailable or undesirable (e.g., due to runtime or external tool dependence), a uniform channel is used in which fixed probabilities are applied to all peptides:

$$p_{\text{loss}} = 0.01, \quad p_{\text{mut}} = 0.005, \quad p_{\text{ins}} = 0.005, \quad p_{\text{shuf}} = 0.005.$$

This parameterization is synthetic and sequence-independent; it provides a stable baseline in which all encoding schemes experience the same residue-level error rates.

Implications for decoding. Residue deletion and insertion change the observed peptide length and can therefore violate downstream assumptions about fixed framing and symbol alignment. In the evaluation pipeline, decoding stages expect peptides to be interpretable under a target length L and a deterministic mapping to an internal symbol/byte representation (Section 3.3.3). Consequently, length violations must be handled explicitly, either by marking affected peptides as erasures or by applying a deterministic normalization procedure (Section 3.6.1).

Beyond length changes, deletions introduce positional drift: removing one residue shifts the positions of all subsequent residues within the peptide. This can desynchronize the mapping from residues to payload bits and induces structured, burst-like corruption in any downstream representation that assumes fixed positions. As a result, a single deletion may affect multiple subsequent bits or symbols in parallel, increasing the effective difficulty for any outer redundancy mechanism operating on fixed-size symbols under limited redundancy.

Algorithm 1 Residue deletion operator (per-position Bernoulli drops with optional empty-peptide removal)

Require: Peptide list $\mathbf{P} = (\pi^{(1)}, \ldots, \pi^{(N)})$, deletion probability p_{loss}, flag $b_{\text{dropEmpty}}$
Ensure: Observed peptide list $\tilde{\mathbf{P}}$

$\tilde{\mathbf{P}} \leftarrow \langle\,\rangle$
for $j \leftarrow 1$ to N **do**
  Let $\pi^{(j)} = (p_1^{(j)}, \ldots, p_L^{(j)})$ with $p_i^{(j)} \in \Sigma$
  $\tilde{\pi} \leftarrow \langle\,\rangle$
  for $i \leftarrow 1$ to L **do**
    Draw $u \sim \text{Unif}(0,1)$
    if $u \geq p_{\text{loss}}$ **then**
      Append $p_i^{(j)}$ to $\tilde{\pi}$
    end if
  end for
  if $|\tilde{\pi}| > 0$ **or** $b_{\text{dropEmpty}} = 0$ **then**
    Append $\tilde{\pi}$ to $\tilde{\mathbf{P}}$
  end if
end for
return $\tilde{\mathbf{P}}$

Algorithm 2 Residue substitution operator (uniform replacement excluding current residue)

Require: Peptide list $\mathbf{P} = (\pi^{(1)}, \ldots, \pi^{(N)})$, alphabet Σ, substitution probability p_{mut}
Ensure: Observed peptide list $\tilde{\mathbf{P}}$

$\tilde{\mathbf{P}} \leftarrow \langle\,\rangle$
for $j \leftarrow 1$ to N **do**
  Let $\pi^{(j)} = (p_1^{(j)}, \ldots, p_L^{(j)})$ with $p_i^{(j)} \in \Sigma$
  $\tilde{\pi} \leftarrow \pi^{(j)}$
  for $i \leftarrow 1$ to L **do**
    Draw $u \sim \text{Unif}(0,1)$
    if $u < p_{\text{mut}}$ **then**
      $\Sigma_i \leftarrow \Sigma \setminus \{p_i^{(j)}\}$
      if $|\Sigma_i| > 0$ **then**
        Draw $a \sim \text{Unif}(\Sigma_i)$
        Set $\tilde{p}_i^{(j)} \leftarrow a$
      end if
    end if
  end for
  Append $\tilde{\pi}$ to $\tilde{\mathbf{P}}$
end for
return $\tilde{\mathbf{P}}$

Algorithm 3 Residue insertion operator (independent insertions with random before/after placement)

Require: Peptide list $\mathbf{P} = (\pi^{(1)}, \ldots, \pi^{(N)})$, alphabet Σ, insertion probability p_{ins}
Ensure: Observed peptide list $\tilde{\mathbf{P}}$

$\tilde{\mathbf{P}} \leftarrow \langle\,\rangle$
for $j \leftarrow 1$ to N **do**
  Let $\pi^{(j)} = (p_1^{(j)}, \ldots, p_L^{(j)})$ with $p_i^{(j)} \in \Sigma$
  if $L = 0$ **then**
    Append $\pi^{(j)}$ to $\tilde{\mathbf{P}}$ ▷ empty peptides are left unchanged
    continue
  end if
  $\tilde{\pi} \leftarrow \langle\,\rangle$
  for $i \leftarrow 1$ to L **do**
    Draw $u \sim \text{Unif}(0,1)$
    if $u < p_{\text{ins}}$ **then**
      Draw $a \sim \text{Unif}(\Sigma)$
      Draw $v \sim \text{Unif}(0,1)$
      if $v < 0.5$ **then**
        Append a to $\tilde{\pi}$ ▷ insert *before*
        Append $p_i^{(j)}$ to $\tilde{\pi}$
      else
        Append $p_i^{(j)}$ to $\tilde{\pi}$
        Append a to $\tilde{\pi}$ ▷ insert *after*
      end if
    else
      Append $p_i^{(j)}$ to $\tilde{\pi}$ ▷ no insertion
    end if
  end for
  Append $\tilde{\pi}$ to $\tilde{\mathbf{P}}$
end for
return $\tilde{\mathbf{P}}$

Algorithm 4 Local shuffle operator

Require: Peptide list $\mathbf{P} = (\pi^{(1)}, \ldots, \pi^{(N)})$, shuffle probability p_{shuf}
Ensure: Observed peptide list $\tilde{\mathbf{P}}$

$\tilde{\mathbf{P}} \leftarrow \langle\,\rangle$
for $j \leftarrow 1$ to N **do**
  Let $\pi^{(j)} = (p_1^{(j)}, \ldots, p_L^{(j)})$ with $p_i^{(j)} \in \Sigma$
  if $L \leq 1$ **or** $p_{\text{shuf}} \leq 0$ **then**
    Append $\pi^{(j)}$ to $\tilde{\mathbf{P}}$
    continue
  end if
  $\tilde{\pi} \leftarrow \pi^{(j)}$
  for $i \leftarrow 1$ to $L - 1$ **do**
    Draw $u \sim \text{Unif}(0, 1)$
    if $u < p_{\text{shuf}}$ **then**
      swap $(\tilde{p}_i^{(j)}, \tilde{p}_{i+1}^{(j)}) \leftarrow (\tilde{p}_{i+1}^{(j)}, \tilde{p}_i^{(j)})$
    end if
  end for
  Append $\tilde{\pi}$ to $\tilde{\mathbf{P}}$
end for
return $\tilde{\mathbf{P}}$

3.6 Decoding and Erasure Handling

Let $\tilde{\mathcal{P}}$ denote the set (or sequence) of observed peptides after channel application, where some peptides may be missing, corrupted, shortened, or contain insertions. Decoding reconstructs an estimate $\hat{\mathbf{x}}$ of the original byte sequence $\mathbf{x}$ by (i) validating and normalizing received peptides, (ii) applying the configured outer redundancy mechanism (RS or fountain), and (iii) reconstructing the scheme payload bitstream and applying the corresponding inverse source/codec stage (Huffman or Yin–Yang).

3.6.1 Peptide normalization, placement, and erasure rules

Decoding assumes that structural metadata produced at encoding time is available (Section 3.1.1), including the target peptide length L, optional index-prefix settings, and the block structure required by the outer redundancy mechanism.

If an index prefix is enabled (Section 3.3.3), each received data peptide is assigned to its intended position by parsing the index prefix. Missing indices are treated as erasures. If indexing is not enabled, the received peptides are assumed to be in the encoder-produced order (or in the order induced by the experimental run configuration).

Erasure criteria. A peptide is marked as an erasure if at least one of the following conditions holds:

- the peptide is missing (no observation for an expected position);
- the peptide is empty;
- the peptide contains symbols outside the restricted alphabet Σ;
- the observed length exceeds the target length ($|\pi| > L$);
- indexing is enabled and the index prefix cannot be parsed or is inconsistent with the expected placement.

Erasures are communicated to the outer decoder at the peptide granularity.

Peptides shorter than the target length ($|\pi| < L$) may arise from residue deletions (Section 3.5). Shortened peptides are not discarded by default. Instead, they are deterministically mapped to the fixed-size internal representation required by the outer decoder by interpreting each residue under the 3-bit mapping ϕ^{-1} (Table 3.1), concatenating the resulting 3-bit symbols, and applying deterministic zero-padding to reach the required fixed length prior to byte packing (Section 3.3.3). Per-peptide length information retained from encoding is used after outer decoding to trim reconstructed data peptides back to their original residue lengths before payload concatenation. This policy preserves peptide-to-position alignment for block-based decoding.

3.6.2 Outer redundancy decoding

Two outer decoding modes are supported by the evaluation design: peptide-level Reed–Solomon decoding and fountain-style peeling decoding. The active mode depends on the evaluated configuration.

For RS-protected configurations, peptides are decoded block-by-block according to the stored block structure. Each peptide is mapped to a fixed-length byte vector of length B (Section 3.3.3), and each RS block is represented as a matrix

$$\hat{\mathbf{M}} \in \{0, \ldots, 255\}^{n \times B},$$

where each row corresponds to one peptide (data or parity) and each column corresponds to one byte position in the packed representation.

RS decoding is performed *column-wise*: for each column $j \in \{1, \ldots, B\}$, an RS decoder over GF(2^8) is invoked on the length-n symbol vector $(\hat{M}_{1,j}, \ldots, \hat{M}_{n,j})$ together with the set of erased row indices from Section 3.6.1. In the standard error/erasure model, column decoding is feasible whenever $2e + s \leq r$, where e denotes the number of unknown symbol errors and s the number of erased symbols given r parity symbols.

After decoding all columns, corrected columns are reassembled into corrected peptide byte vectors and mapped back to peptide sequences over Σ. Parity peptides are then discarded, yielding a corrected stream of *data* peptides.

If interleaving is enabled (Section 3.4.1), RS decoding is performed in the interleaved domain, and the inverse interleaving permutation is applied *after* RS decoding to restore the original data peptide order for payload reconstruction.

3.6.3 Fountain decoding: CRC filtering and LT peeling

For fountain-style configurations, decoding is rateless and proceeds by collecting and processing received droplets until the k_{F} source symbols can be recovered. The observed peptide stream is first converted back to a bytestream using the same inverse residue packing used elsewhere in the pipeline (Section 3.6.4). The bytestream is then segmented into fixed-length droplet packets using the configured droplet packet length. Any incomplete trailing packet is ignored.

Each reconstructed droplet packet is validated using its CRC. Packets that fail CRC validation are discarded and do not contribute equations, effectively converting residue-level corruption into droplet erasures. For each remaining packet, the decoder parses the header (seed and degree) and rebuilds the same subset of source-symbol indices that was used at encoding time. Degree-1 packets directly identify a single source symbol; for higher degrees, the decoder replays the seed-initialized pseudo-random selection to obtain the set of distinct indices.

Each valid droplet defines one XOR equation over the unknown source symbols: the droplet body equals the bytewise XOR of the source symbols referenced by its rebuilt index set. Collecting all valid droplets yields a sparse system of XOR constraints, where sparsity is induced by the degree distribution.

Decoding applies iterative peeling. All degree-1 equations are inserted into a work queue. When a degree-1 equation reveals a source symbol, that symbol is recorded and XOR-eliminated from every other equation that contains it, reducing their degrees. Any equation that becomes degree-1 due to elimination is added to the queue. This process continues until either

(i) all k_{F} source symbols are recovered, or (ii) the queue becomes empty, indicating that decoding has stalled due to insufficient or inconsistent equations.

If all source symbols are recovered, they are concatenated in index order and the result is truncated to the original file length (available as decoding metadata) to obtain the reconstructed byte sequence $\hat{\mathbf{x}}$. If peeling stalls before full recovery, the run is labeled as an outer-decoder failure under the criteria in Section 3.7.

3.6.4 Payload reconstruction and inverse source decoding

After outer decoding, corrected *data* peptides are converted back to the payload bitstream. If an index prefix is enabled (Section 3.3.3), the prefix residues are removed prior to payload concatenation. Residues are mapped back to 3-bit symbols using ϕ^{-1} and concatenated in order. Alignment padding bits added during encoding to reach a multiple of 3 are removed using the recorded padding metadata (Section 3.3.3).

For the Huffman-baseline configuration, the recovered payload bitstream is decoded using the corresponding Huffman codebook to obtain $\hat{\mathbf{x}}$, which is compared to the original bytes $\mathbf{x}$ according to the success criterion in Section 3.1.1. For the Yin–Yang configuration, the recovered bitstream is passed through the inverse constrained mapping to reconstruct $\hat{\mathbf{x}}$. For fountain configurations , recovered source symbols are concatenated and truncated to the recorded original file length to yield $\hat{\mathbf{x}}$.

3.7 Evaluation Protocol and Metrics

This section defines the metrics computed from each run and the minimal counters recorded in the run log so that all reported quantities can be derived automatically. A run yields an original byte sequence $\mathbf{x}$ and a decoded byte sequence $\hat{\mathbf{x}}$ (Section 3.2.2). In addition, the encoder records run-level transmission counters such as the number of transmitted units and the total number of transmitted residues.

3.7.1 Comparison criterion

Across encoding schemes and redundancy settings, the primary objective is end-to-end reliability under identical channel conditions. Configurations are therefore compared first by perfect recovery rate (PRR). If two configurations reach comparable PRR on the same test set and channel, the tie is broken by transmitted overhead in bytes (Δ_{enc}); lower overhead is preferred. Secondary indicators (redundancy ratio, net rate, and utilization) are reported to

explain where overhead originates (parity or droplet count, index prefixes, and padding), but they do not override the primary ordering by PRR and Δ_{enc}.

3.7.2 Evaluation metrics

Each run produces an original byte sequence $\mathbf{x}$ and a decoded byte sequence $\hat{\mathbf{x}}$. Let $m = |\mathbf{x}|$ and $\hat{m} = |\hat{\mathbf{x}}|$ denote the corresponding lengths in bytes.

Success is a strict indicator that is 1 only if the decoded output matches the original input byte-for-byte:

$$\text{success} \;=\; \mathbf{1}[\hat{\mathbf{x}} = \mathbf{x}].$$

Perfect recovery rate (PRR) reports how often perfect recovery occurs within a set of runs $\mathcal{R}$:

$$\text{PRR} \;=\; \frac{1}{|\mathcal{R}|} \sum_{r \in \mathcal{R}} \text{success}(r).$$

Length deviation quantifies whether decoding produces an output that is shorter or longer than the original:

$$\Delta_m \;=\; \hat{m} - m,$$

reported together with the magnitude $|\Delta_m|$.

When $m \neq \hat{m}$, only the first $m_{\min}$ bytes exist in both sequences and can be compared position-wise:

$$m_{\min} \;=\; \min(m, \hat{m}).$$

Byte error count measures how many byte positions differ in the comparable prefix, plus a penalty for missing or extra bytes:

$$E_{\text{byte}} \;=\; \sum_{i=1}^{m_{\min}} \mathbf{1}[x_i \neq \hat{x}_i] \;+\; |m - \hat{m}|.$$

Bit error count measures the number of incorrect bits in the comparable prefix (using per-byte Hamming distance) and adds a bit penalty for missing or extra bytes:

$$E_{\text{bit}} \;=\; \sum_{i=1}^{m_{\min}} \text{hd}_8(x_i, \hat{x}_i) \;+\; 8|m - \hat{m}|,$$

where $\text{hd}_8(\cdot, \cdot)$ denotes Hamming distance between the 8-bit representations of two bytes.

Bit error rate (BER) normalizes the bit error count by the original bit length so runs of different sizes are comparable:

$$\mathrm{BER} = \frac{E_{\mathrm{bit}}}{8m}.$$

Over a run set $\mathcal{R}$, the mean BER is reported as:

$$\overline{\mathrm{BER}} = \frac{1}{|\mathcal{R}|} \sum_{r \in \mathcal{R}} \mathrm{BER}(r).$$

Optionally, a byte-level error rate can be reported as $\mathrm{ByteER} = E_{\mathrm{byte}}/m$.

Failure categorization. Each run is assigned a **failure label** to distinguish typical failure modes during analysis:

- **success** if $\hat{\mathbf{x}} = \mathbf{x}$;
- **empty output** if $\hat{m} = 0$;
- **outer-decoder failure** if the configured outer stage signals failure (e.g., RS decoding failure; fountain peeling stalls);
- **source/codec failure** if the inverse source stage signals failure (e.g., invalid compressed representation);
- **mismatch** otherwise (non-empty output with $\hat{\mathbf{x}} \neq \mathbf{x}$ and no explicit failure signal).

The following metrics are computed from transmission counters recorded during encoding. Let N_{data} be the number of source data units (data peptides for RS-protected configurations; source packets for fountain configurations) and N_{tx} the total number of transmitted units (data plus parity peptides for RS; transmitted droplets for fountain). The **redundancy ratio** is:

$$\mathrm{RR} = \frac{N_{\mathrm{tx}}}{N_{\mathrm{data}}}.$$

Let R_{tot} denote the total number of transmitted amino-acid residues in the run (including parity units and index prefixes when present). The **net information rate (bits per amino acid)** is:

$$\mathrm{b/AA} = \frac{8m}{R_{\mathrm{tot}}}.$$

To express the residue-level transmission cost in absolute byte units, define the **byte-equivalent transmitted size** assuming $|\Sigma| = 8$ (i.e., 3 bits per residue):

$$\ell_{\text{enc}} = \left\lceil \frac{3R_{\text{tot}}}{8} \right\rceil,$$

and the **encoded-size overhead** relative to the original file size:

$$\Delta_{\text{enc}} = \ell_{\text{enc}} - m.$$

Finally, let C_{tot} be the total available payload capacity (in bits) across all transmitted peptides under the configured payload length, and let b_{use} be the number of useful payload bits before tail alignment padding. The **payload utilization** is:

$$\mathrm{U} = \frac{b_{\text{use}}}{C_{\text{tot}}}.$$

Table 3.3 summarizes the metrics reported throughout the evaluation.

Table 3.3: Summary of evaluation metrics.

Metric	Definition
success	$\mathbf{1}[\hat{\mathbf{x}} = \mathbf{x}]$
PRR	$\lvert\mathcal{R}\rvert^{-1} \sum_{r\in\mathcal{R}} \text{success}(r)$
Δ_m	$\lvert\hat{\mathbf{x}}\rvert - \lvert\mathbf{x}\rvert$ (also report $\lvert\Delta_m\rvert$)
E_{byte}	$\sum_{i=1}^{m_{\min}} \mathbf{1}[x_i \neq \hat{x}_i] + \lvert m - \hat{m}\rvert$, where $m_{\min} = \min(\lvert\mathbf{x}\rvert, \lvert\hat{\mathbf{x}}\rvert)$
E_{bit}	$\sum_{i=1}^{m_{\min}} \text{hd}_8(x_i, \hat{x}_i) + 8\lvert m - \hat{m}\rvert$
BER	$E_{\text{bit}}/(8\lvert\mathbf{x}\rvert)$ (per run)
$\overline{\text{BER}}$	$\lvert\mathcal{R}\rvert^{-1} \sum_{r\in\mathcal{R}} \text{BER}(r)$ (over a run set)
Failure label	{`success`, `mismatch`, ...}
RR	$N_{\text{tx}}/N_{\text{data}}$
b/AA	$8\lvert\mathbf{x}\rvert/R_{\text{tot}}$
ℓ_{enc}	$\lceil 3R_{\text{tot}}/8\rceil$ (equivalent transmitted size in bytes)
Δ_{enc}	$\ell_{\text{enc}} - \lvert\mathbf{x}\rvert$
U	$b_{\text{use}}/C_{\text{tot}}$

4 Results

Results are organized by scheme family to keep the interpretation aligned with the design choices in the pipeline.

4.1 Huffman baseline

The Huffman baseline serves as the reference configuration for end-to-end behavior under the shared peptide pipeline. The first goal is to determine an RS parity profile that yields a stable operating point. This section therefore separates a conservative *worst-case* sweep (uniform, sequence-agnostic errors) from the more realistic *score-driven* channel (sequence-aware errors), which is used for the main scheme comparisons.

4.1.1 Worst-case uniform sweep: upper-bound parity requirement

The first sweep is executed under the *uniform, sequence-agnostic* channel parameterization (Section 3.5.1). The fixed probabilities are intentionally chosen on the high side to act as a conservative stress test: all peptides experience the same perturbation strength, independent of sequence content. The purpose is not to model realistic synthesis variability, but to determine an RS setting that remains stable under adverse conditions.

Table 4.1 summarizes the sweep by aggregating per-run outcomes by ECC profile. Under this stress test, lower-parity profiles show substantial residual discrepancies and frequent failures for larger files, while increasing parity improves stability across the evaluated size range. In this setting, RS128 achieves perfect recovery across the sweep and serves as an *upper-bound* operating point for worst-case robustness.

Table 4.1: Huffman baseline: summary of the RS parity sweep under the uniform (sequence-agnostic) channel (mean over file sizes up to 1 MB).

ECC profile	PRR	Mean BER	Mean $\lvert\Delta_m\rvert$ (bytes)
RS8	0.3333	0.533678	99435
RS16	0.3810	0.534096	99508
RS32	0.5238	0.387583	97745
RS64	0.8095	0.068165	18564
RS128	1.0000	0.000000	0

Figure 4.1 isolates RS128 for the same sweep. Across the evaluated file-size range, the BER remains zero, consistent with the perfect recovery rate reported in Table 4.1. This motivates using RS128 as a conservative reference point when interpreting later results.

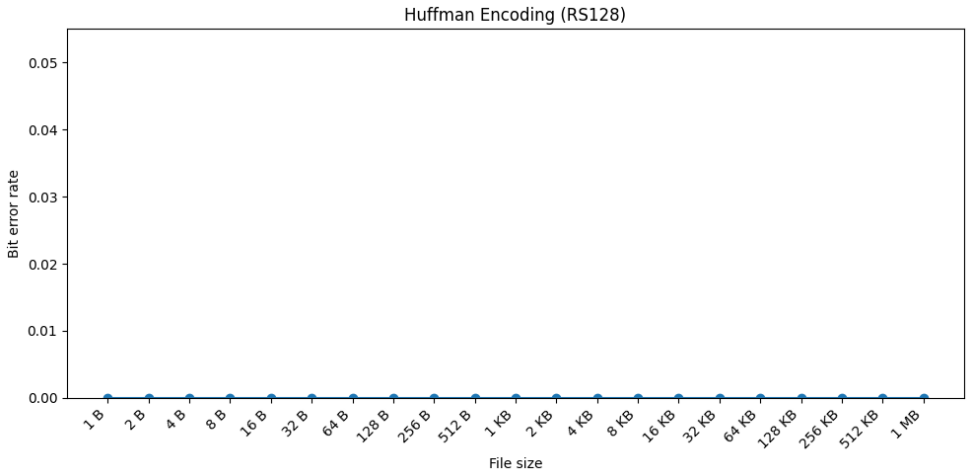


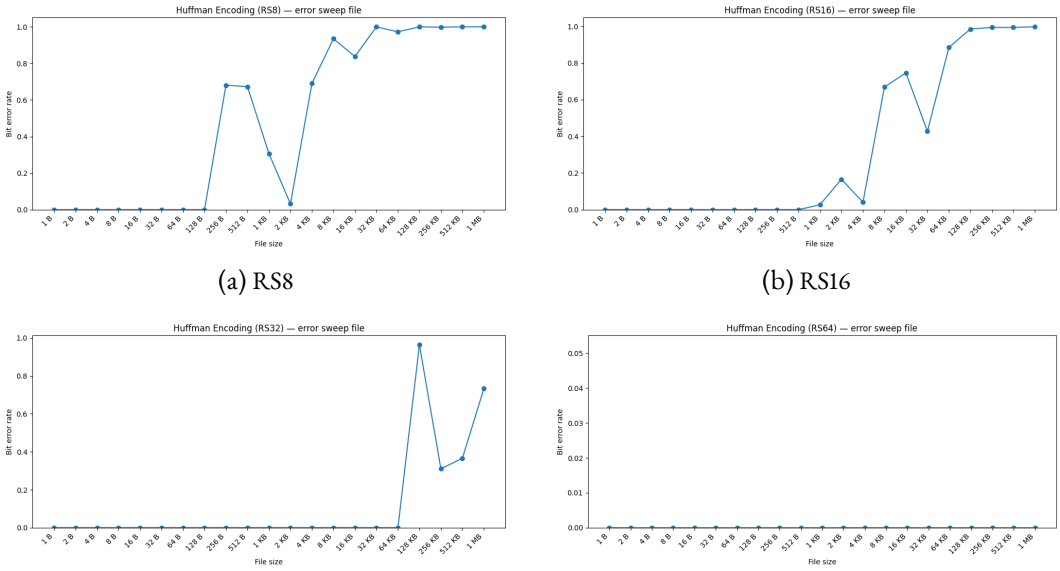
Figure 4.1: Huffman baseline under the uniform (sequence-agnostic) channel: RS128 mean BER vs. file size.

4.1.2 Score-driven channel: reduced parity suffices

The main comparisons in the remainder of the thesis use the score-driven channel parameterization (Section 3.5.1). In this setting, peptides with higher predicted quality receive lower corruption rates, which reduces the effective error burden relative to the uniform stress test and better reflects heterogeneous synthesis difficulty.

Figure 4.2 shows the corresponding parity sweep under the score-driven channel. RS8 and RS16 still exhibit a strong size dependence, with a transition from low BER at small sizes to near-random outputs at larger sizes. RS32 improves robustness but remains unstable for

large files. In contrast, RS64 exhibits zero BER across the evaluated size range in this sweep, indicating stable end-to-end recovery under the sequence-aware model.



(a) RS8 (b) RS16

(c) RS32 (d) RS64

Figure 4.2: Huffman baseline under the score-driven channel: mean BER vs. file size for multiple RS parity profiles.

To quantify this trade-off numerically, Table 4.2 summarizes the same sweep by aggregating per-run outcomes by ECC profile.

Table 4.2: Huffman baseline: score-driven channel summary.

ECC profile	PRR	Mean BER	Mean $\lvert\Delta_m\rvert$ (bytes)
RS8	0.3810	0.481922	99141
RS16	0.4762	0.330021	96641
RS32	0.8095	0.113034	42619
RS64	1.0000	0.000000	0

Selected configuration for subsequent comparisons

The uniform stress test indicates that RS128 is required as an upper-bound setting for worst-case robustness. However, under the score-driven channel used for the main evaluation, RS64 already achieves stable recovery across the tested file sizes while reducing transmitted redundancy relative to RS128. Therefore, RS64 is used as the default Huffman operating point

for the subsequent cross-scheme comparisons under the score-driven model, while RS128 is retained as a conservative reference for worst-case interpretation.

To make the overhead at the selected operating point explicit, Table 4.3 reports representative sizes and the corresponding transmitted-size overhead (in bytes) implied by the run-level counters.

Table 4.3: Huffman baseline with RS64 (score-driven): representative overhead indicators.

File size (bytes)	$\mathcal{L}_{\text{enc}}$ **(bytes)**	Δ_{enc} **(bytes)**	**RR**	**b/AA**	U
1	439	438	65.00	0.007	0.148
128	527	399	5.57	0.729	0.995
1024	2957	1933	3.71	1.039	0.998
16384	47426	31042	3.69	1.036	1.000
262144	757263	495119	3.67	1.039	1.000
1048576	3029400	1980824	3.67	1.038	1.000

4.2 Yin–Yang constrained mapping

The RS operating point is taken over from the Huffman baseline for two reasons. First, the outer RS layer is *mapping-agnostic* in the sense that it operates on the protected byte stream regardless of whether the inner stage is Huffman or Yin–Yang; hence selecting a stable RS profile for the score-driven channel yields a comparable redundancy setting across schemes. Second, under the score-driven channel the effective error burden depends on the sequence distribution, but RS64 already provides a strong baseline that decodes reliably across the full tested file-size range.

Empirically, Yin–Yang with RS64 achieves perfect recovery across all evaluated file sizes (PRR = 1.0000 with mean BER = 0 and mean $|\Delta_m| = 0$)

To make the transmitted overhead of the RS64 operating point explicit, Table 4.4 reports representative overhead indicators derived from the run-level transmission counters.

Table 4.4: Yin–Yang mapping with RS64 (score-driven): representative overhead indicators.

File size (bytes)	$\mathcal{L}_{\text{enc}}$ (bytes)	Δ_{enc} (bytes)	RR	b/AA	U
1	803	802	65.00	0.030	0.222
128	1060	932	5.41	0.362	0.981
1024	7734	6710	3.71	0.396	0.998
16384	119400	103016	3.68	0.388	1.000
262144	1899120	1636976	3.67	0.388	1.000
1048576	4803290	3754714	3.67	0.388	1.000

To characterize how Yin–Yang behaves with a reduced parity budget, the same file-size sweep is repeated at RS32. Across the evaluated sizes, Yin–Yang attains PRR = 0.9524 (20/21 successful runs), with essentially vanishing residual discrepancy (mean BER 1.0×10^{-7} and mean $|\Delta_m| = 0$). The single failure occurs at the largest file size (1 MB) and manifests as a small mismatch rather than a full breakdown into a near-random output.

These results are consistent with the intended effect of the constrained mapping: by avoiding unfavorable residue patterns, the encoder shifts the generated sequence distribution toward higher predicted scores and lower induced base-error rates, reducing the effective error burden under the score-driven channel at a fixed parity budget.

4.3 Huffman vs. Yin–Yang at matched RS budgets

To isolate the effect of the inner mapping, Huffman and Yin–Yang are compared at the same RS profiles under the score-driven channel. At RS64 both schemes achieve perfect recovery, so RS64 mainly reflects efficiency at a saturated reliability point. RS32 is more discriminative because the system operates closer to the decoding threshold, where mapping choices affect how the score-driven channel corrupts the generated peptides.

Table 4.5: Huffman vs. Yin–Yang under the score-driven channel at RS64.

Scheme	PRR	Mean BER	Mean $\lvert\Delta_m\rvert$	Δ_{enc} @ 1MB	RR @ 1MB	b/AA @ 1MB
Huffman	1.0000	0	0	1 980 824	3.667	1.038
Yin–Yang	1.0000	0	0	3 754 714	3.667	0.388

At RS64, both mappings saturate at PRR = 1, so recovery metrics no longer distinguish them. The remaining difference is efficiency: in this implementation, Yin–Yang incurs a

larger transmitted-size overhead and a lower net information rate (b/AA), i.e., it spends more residues per recovered payload bit at the same RS profile.

Table 4.6: Huffman vs. Yin–Yang under the score-driven channel at RS32.

Scheme	PRR	Mean BER	Mean $\lvert\Delta_m\rvert$	Δ_{enc} @ 1MB	RR @ 1MB	b/AA @ 1MB
Huffman	0.8095	0.113034	42 619	879 224	2.333	1.632
Yin–Yang	0.9524	1.02×10^{-7}	0	2 621 649	2.333	0.857

At RS32, the mappings separate clearly: Yin–Yang maintains substantially higher recovery (PRR) with essentially zero residual BER and no length deviation on average, whereas Huffman shows more frequent failures and higher residual discrepancy. This is the regime where the score-driven channel makes the inner mapping relevant: constraint-based mapping can shift the encoded sequence distribution toward higher predicted quality, reducing the effective error burden at the same parity budget. In exchange, Yin–Yang shows higher transmitted overhead and lower b/AA in the current codec implementation.

4.4 Fountain coding

In addition to RS-based outer protection, the pipeline includes a fountain-style configuration with per-droplet integrity filtering (CRC) and peeling decoding. Because droplets that fail the CRC are discarded before solving, the effective error profile seen by the fountain decoder differs qualitatively from the RS setting. For this reason, fountain results are reported as a feasibility and scaling study, and redundancy is analyzed along the droplet-overhead axis rather than via an RS parity profile.

Fountain redundancy is parameterized by an overhead factor ρ, where the encoder targets $(1 + \rho)\,k$ transmitted droplets for k source symbols. In the implementation and run logs, a profile `FNTXX` encodes $\rho = \texttt{XX}/10$. For example, `FNT50` corresponds to $\rho = 5.0$ (i.e., $\approx 6k$ transmitted droplets).

Unlike the RS comparison runs, fountain evaluation is performed only under the uniform channel setting (Section 3.5), because the present fountain pipeline converts many corruptions into droplet erasures via CRC filtering and does not expose a score-driven per-peptide parameterization that is directly comparable to the RS runs. The uniform channel probabilities are:

$$p_{\text{loss}} = 0.01, \qquad p_{\text{mut}} = p_{\text{ins}} = p_{\text{shuf}} = 0.005.$$

To identify a conservative operating point (analogous to the Huffman upper-bound sweep), a redundancy sweep is performed over $\rho \in \{1.0, 2.0, 3.0, 5.0\}$ (profiles `FNT10`, `FNT20`, `FNT30`, `FNT50`) across the full file-size sweep up to 1 MB.

Table 4.7: Fountain redundancy sweep (uniform channel): aggregated recovery metrics over the full file-size sweep (up to 1 MB).

Profile	ρ	PRR	Mean BER	Failures (file sizes)
FNT10	1.0	0.2857	0.714286	≥ 64 B
FNT20	2.0	0.3810	0.619048	≥ 256 B
FNT30	3.0	0.9048	0.095238	512 B, 1 024 B
FNT50	5.0	1.0000	0.000000	–

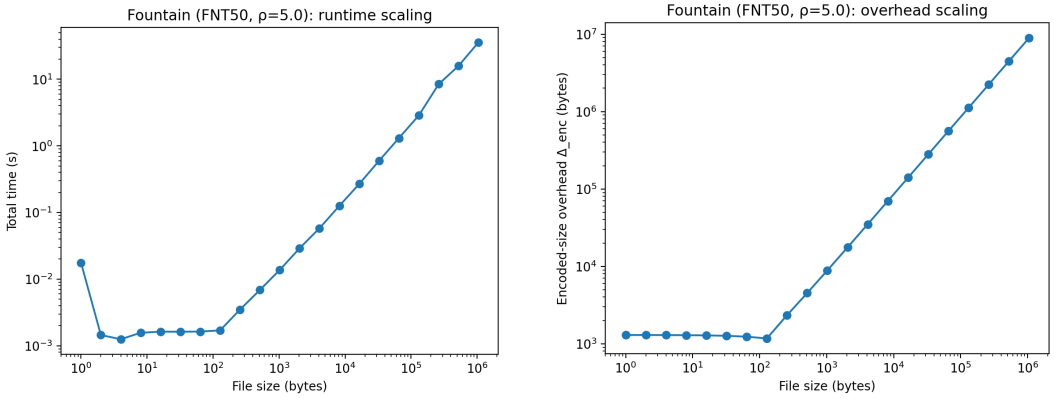
Over the full sweep, `FNT50` ($\rho = 5.0$) is the smallest evaluated profile that achieves perfect recovery (PRR= 1) for all tested file sizes. `FNT30` ($\rho = 3.0$) is already sufficient for the large-file regime, but exhibits failures at intermediate small sizes (512 and 1 024 bytes) under the enforced minimum-droplet constraints and the resulting peeling dynamics. Therefore, `FNT50` is used as a conservative reference setting for reporting overhead and scaling indicators below.

Table 4.8 reports representative operating points derived from transmission counters. For very small files, a minimum droplet budget dominates and yields disproportionately high overhead; for larger files, the redundancy ratio approaches the intended value implied by ρ (i.e., $1 + \rho = 6$).

Table 4.8: Fountain configuration (`FNT50`, $\rho = 5.0$): representative overhead and runtime indicators.

File size (bytes)	$\mathcal{L}_{\text{enc}}$ (bytes)	Δ_{enc} (bytes)	RR	b/AA	U	Total time (s)
1	1 296	1 295	48.00	0.002315	0.000772	0.018
128	1 296	1 168	6.00	0.296296	0.098765	0.002
1 024	9 882	8 858	6.00	0.310868	0.103623	0.014
16 384	156 168	139 784	6.00	0.314738	0.104913	0.268
262 144	2 498 202	2 236 058	6.00	0.314799	0.104933	8.452
1 048 576	9 992 322	8 943 746	6.00	0.314815	0.104938	35.776

Scaling behavior. While `FNT50` reaches PRR= 1 under this uniform setting, runtime increases steeply with file size, with peeling/solving dominating at the largest instances.



(a) Total runtime vs. file size (log scales).

(b) Encoded-size overhead vs. file size (log scales).

Figure 4.3: Fountain configuration (`FNT50`, $\rho = 5.0$): scaling behavior across the file-size sweep.

A direct comparison to the RS results in Sections 4.1–4.3 requires matching the comparison axis. In particular, fountain decoding converts many corruptions into erasures via CRC filtering, and performance is primarily controlled by droplet overhead rather than parity count. A fair head-to-head would therefore evaluate PRR as a function of transmitted overhead (Δ_{enc}, RR, or b/AA) under matched channel assumptions.

5 Discussion

This chapter interprets the empirical findings of Chapter 4 with respect to the research question introduced in Section 3.1.1 and the intended scope set in the Introduction. The discussion emphasizes (i) transferability of DNA-oriented coding ideas to peptide sequences under minimal adaptation, (ii) the role of the adopted channel model in determining which design choices matter, and (iii) the practical trade-offs between reliability, redundancy, and efficiency that emerge within the implemented pipeline.

5.1 Interpretation of results and answer to the research question

The research question asks whether coding schemes originally developed for DNA-based molecular storage can be applied to peptides without *fundamental changes*, where fundamental changes refer to redesigning the core algorithmic logic rather than adapting alphabet mapping, framing, constraint handling, or parameters. Within the implemented pipeline and the reported experiments, the empirical answer is twofold.

First, **transferability holds at the level of core principles**. A Huffman-based baseline, a Yin–Yang-style constrained mapping, and an LT fountain construction can all be instantiated over fixed-length peptide sequences using only minimal adaptations (alphabet restriction and mapping, fixed peptide framing, and parameter tuning), while preserving the conceptual logic of each scheme family. In this sense, the study supports the claim that DNA-oriented coding ideas can be carried over to peptides without re-engineering their central mechanisms.

Second, **peptide-specific considerations become necessary for meaningful comparisons**. The results show that the impact of inner mapping choices depends critically on how the peptide channel is modeled. Under a sequence-agnostic channel, constrained mapping has no mechanism to reduce corruption, so performance is dominated by the outer-code budget. Under a sequence-aware channel, constrained mapping can matter because it can steer the sequence distribution toward higher predicted quality and thereby reduce the effective corruption burden seen by the outer decoder. Consequently, while transfer is feasible,

claims of mapping-driven improvement are only meaningful under an error model that reflects peptide-specific sequence dependence.

This dependency is already visible from the Huffman baseline parity sweeps. The uniform stress test (Table 4.1) is evaluated over multiple independent channel realizations to obtain a conservative reference point. It establishes an upper bound on parity requirements under sequence-agnostic errors: increasing RS parity improves stability, and RS128 reaches perfect recovery across the sweep (PRR = 1). This reference is useful because it is independent of any sequence-aware effects; it primarily reflects the interaction between residue-level perturbations, fixed framing, and block-based outer decoding.

Under the score-driven channel, the baseline additionally identifies a stable operating point for cross-scheme comparisons within the same channel setting. Table 4.2 and Figure 4.2 show that RS64 already achieves perfect recovery across file sizes up to 1 MB, motivating RS64 as the default operating point for the main comparisons. Importantly, this illustrates the central modeling implication: once the channel assigns lower error rates to high-quality peptides, substantially less parity is required than under the uniform stress test.

At this selected operating point, Huffman also establishes an efficiency reference. At RS64, Huffman attains a comparatively high net information rate (Table 4.3) because it uses the full 3-bit residue mapping at the payload layer and does not reduce the payload rate to gain additional degrees of freedom. This becomes the baseline for interpreting constrained mappings that explicitly trade rate for sequence selection.

The Yin–Yang configuration introduces such selection freedom by mapping each 2-bit payload symbol to a pair of residues and choosing the emitted residue to avoid undesirable local patterns. Crucially, this does not change the outer-code structure: RS acts on the serialized stream regardless of the inner mapping. Therefore, the empirical advantage of Yin–Yang is expected to appear primarily near the decoding threshold, where a reduction in effective corruption can change whether the outer decoder succeeds.

The results support this interpretation. At RS32, Table 4.6 shows a clear separation: Yin–Yang achieves higher PRR than Huffman with essentially vanishing residual discrepancy, whereas Huffman exhibits more failures and non-zero BER. This is precisely the regime in which a *sequence-aware* channel makes inner mapping relevant: if the channel penalizes low-quality sequences, then a constrained mapper can reduce the effective corruption burden by shifting outputs toward higher predicted quality. At RS64, however, both mappings reach PRR = 1 (Table 4.5), so reliability no longer discriminates designs; the remaining difference is efficiency. Because Yin–Yang carries only 2 bits per payload residue, it incurs a lower net rate and higher transmitted-size overhead than the 3-bit Huffman baseline, as reflected by the b/AA and Δ_{enc} indicators in Table 4.5. Overall, Yin–Yang is not a universal win: it can

improve robustness at reduced RS budgets under a sequence-aware channel, but it trades off payload efficiency.

Finally, the fountain configuration should be interpreted as operating in a different regime than RS parity sweeps. RS redundancy is controlled by a parity count per block, while fountain redundancy is controlled by transmitted overhead ρ (droplets per source symbol). Moreover, integrity filtering discards invalid droplets before decoding, converting corruption into erasures at the droplet level and altering the effective error profile seen by the solver relative to the RS pipeline. Under the feasibility setting used in Chapter 4, the fountain configuration achieves perfect recovery at $\rho = 5.0$ in the tested uniform error regime and reveals a practical bottleneck: runtime grows steeply with file size (Figure 4.3). Therefore, the fountain results are best interpreted as feasibility plus scaling behavior in the current implementation, rather than as a direct head-to-head winner/loser statement against RS profiles.

5.2 Limitations and practical implications

The conclusions above are conditioned on the evaluation scope and several simplifying assumptions that influence interpretation.

First, the evaluation restricts the alphabet to $|\Sigma| = 8$ and uses fixed peptide framing to maintain a clean 3-bit mapping and comparable interfaces across schemes. The experiments therefore assess transferability under a controlled alphabet restriction rather than the maximum achievable payload density with larger peptide alphabets. Second, channel modeling is decisive: under a sequence-agnostic channel, constrained mapping cannot influence corruption; under a score-driven channel, it can. Hence, claims about the benefit of constrained mapping are inherently tied to the adopted sequence-aware parameterization. Third, beyond the uniform stress test (which uses multiple independent channel draws to obtain a conservative upper-bound reference), the remaining file–configuration points are reported as point estimates under one channel realization. Replicating these points would primarily add confidence intervals for PRR/BER under the score-driven model rather than change the evaluation protocol. In addition, decoding assumes that structural metadata is available externally and is not embedded into the peptide payloads. Finally, the study is computational and does not include wet-lab validation; absolute performance levels should not be interpreted as laboratory guarantees, while comparative statements are valid only under the stated model.

Despite these limitations, the results suggest clear design guidance within the implemented setup. If the goal is **high payload efficiency** at a reliable operating point, the Huffman baseline with a stable RS profile (e.g., RS64 under the score-driven channel) provides high b/AA and lower transmitted overhead at large file sizes (Table 4.3). If the goal is **robustness at**

reduced RS budgets under a sequence-aware channel, constrained mapping can improve PRR near the decoding threshold (e.g., at RS32) but at the cost of a lower net information rate (Table 4.6). If the goal is **rateless operation**, fountain coding is feasible in the pipeline, but runtime scaling becomes a primary constraint at large file sizes in the current implementation (Figure 4.3), and comparisons should be made along matched transmitted-overhead axes rather than parity profiles.

6 Conclusion and Outlook

This thesis examined whether coding strategies developed for DNA-based molecular storage can be applied to peptide-based storage within a unified software evaluation pipeline. The pipeline encodes files into fixed-length peptide sequences over a restricted alphabet, applies a residue-level stochastic channel, and decodes perturbed outputs using either a block-based Reed–Solomon (RS) outer code or a fountain-style peeling decoder. Performance is reported using perfect recovery rate (PRR) together with transmitted overhead and efficiency indicators.

6.1 Conclusion

The main conclusions are:

- **DNA-oriented scheme families transfer to a peptide setting without redesign.** Huffman source coding, Yin–Yang-style constrained mapping, and LT fountain coding can be instantiated over peptide sequences while preserving their core mechanisms. The required adaptations are limited to alphabet mapping, fixed framing, and parameter selection.
- **RS parity yields a robust, mapping-agnostic reliability backbone.** In the uniform stress test evaluated over multiple independent channel draws, increasing parity improves stability and RS128 serves as a conservative upper-bound setting. Under the score-driven channel used for the main comparisons, RS64 already achieves stable recovery across file sizes up to 1 MB.
- **Constrained mapping improves robustness near the RS threshold in a sequence-aware channel.** At RS32, Yin–Yang achieves higher PRR than the Huffman baseline with essentially vanishing residual discrepancy. At RS64, both mappings saturate at perfect recovery, and the remaining difference is efficiency.
- **The main cost of constrained mapping is efficiency.** Because the implemented Yin–Yang payload mapping operates at 2 bits per payload residue, it attains a lower net in-

formation rate and higher transmitted overhead than the 3-bit baseline at comparable reliability.

- **Fountain coding is feasible but operates in a different redundancy regime.** Under the uniform channel, a redundancy sweep indicates that `FNT50` ($\rho = 5.0$) is the smallest tested setting that achieves PRR= 1 across the full file-size sweep up to 1 MB, while lower overhead factors fail for parts of the sweep. Although recovery can be made reliable by increasing droplet overhead and using CRC-based filtering (converting many corruptions into erasures), runtime grows steeply with file size in the current implementation. Comparisons to RS-based results should therefore be made along a transmitted-overhead axis (RR, Δ_{enc}, b/AA) rather than parity profiles.

6.2 Outlook

Several extensions would strengthen the findings and improve realism:

- **Extending repeated-draw evaluation beyond the stress test.** While multiple independent channel draws are already used for the uniform stress test to establish a conservative upper-bound RS setting, replicating the remaining configurations would provide confidence intervals for PRR/BER under the score-driven model and quantify variability around the reported point estimates.
- **Metadata embedding.** Embed essential structural metadata (padding, block structure, indexing) into the peptide payload to remove the external-metadata assumption and quantify the resulting overhead.
- **Larger alphabets and hybrid mappings.** Evaluate expanded peptide alphabets (beyond $|\Sigma| = 8$) and mixed mappings that exploit peptide chemistry while maintaining robust decoding interfaces.
- **Matched-overhead RS vs. fountain evaluation.** Re-run fountain under the same channel parameterization as the RS experiments and report PRR as a function of transmitted overhead (RR, Δ_{enc}, b/AA) to enable head-to-head comparisons.
- **Closer-to-lab channel models.** Incorporate peptide-specific effects such as sequence-dependent hotspots, readout ambiguity, and multi-read consensus to better approximate mass-spectrometry workflows.

Code and Data Availability

The source code of the evaluation pipeline, including encoding/decoding implementations, channel simulation, and experiment scripts used to produce the results in this thesis, aswell as the experiment and outputs (run logs, tables, and plots) are archived on Hessenbox at:

https://next.hessenbox.de/index.php/s/ic7AKqBB3dz6QeP
Password: E:;5Hw@5BbTq'rW

A framework with a front end to provide more details and visualizations of the software used in our thesis :

https://peptidethesis.org

Bibliography

[1] D. Reinsel, J. Gantz, and J. Rydning, "The digitization of the world: From edge to core." White Paper, 2018, international Data Corporation (IDC), Sponsored by Seagate. [Online]. Available: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf

[2] A. Doricchi, C. M. Platnich, A. Gimpel, F. Horn, M. Earle, G. Lanzavecchia, A. L. Cortajarena, L. M. Liz-Marzán, N. Liu, R. Heckel, R. N. Grass, R. Krahne, U. F. Keyser, and D. Garoli, "Emerging approaches to dna data storage: Challenges and prospects." *ACS Nano*, vol. 16, no. 11, pp. 17 552–17 571, 2022, pMID: 36256971. [Online]. Available: https://doi.org/10.1021/acsnano.2c06748

[3] S. Jo, H. Shin, S.-Y. Joe, D. Baek, C. Park, and H. Chun, "Recent progress in DNA data storage based on high-throughput DNA synthesis," *Biomedical Engineering Letters*, vol. 14, no. 5, pp. 993–1009, 2024, pMID: 39220021; PMCID: PMC11362454. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/39220021/

[4] Y. Dong, F. Sun, Z. Ping, Q. Ouyang, and L. Qian, "DNA storage: research landscape and future prospects," *National Science Review*, vol. 7, no. 6, pp. 1092–1107, 2020, pMID: 34692128; PMCID: PMC8288837. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/34692128/

[5] C. Imburgia, L. Organick, K. Zhang, N. Cardozo, J. McBride, C. Bee, D. Wilde, G. Roote, S. Jorgensen, D. Ward, C. Anderson, K. Strauss, L. Ceze, and J. Nivala, "Random access and semantic search in DNA data storage enabled by Cas9 and machine-guided design," *Nature Communications*, vol. 16, no. 1, p. 6388, 2025, pMID: 40640156; PMCID: PMC12246221. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/40640156/

[6] T. Buko, N. Tuczko, and T. Ishikawa, "DNA data storage," *BioTech*, vol. 12, no. 2, p. 44, 2023, pMID: 37366792; PMCID: PMC10296570. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/37366792/

[7] S. Wang, X. Mao, F. Wang, X. Zuo, and C. Fan, "Data storage using DNA," *Advanced Materials*, vol. 36, no. 6, p. e2307499, 2024, pMID: 37800877. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/37800877/

[8] C. C. A. Ng, W. M. Tam, H. Yin, Q. Wu, P.-K. So, M. Y.-M. Wong, F. C. M. Lau, and Z.-P. Yao, "Data storage using peptide sequences." Journal Article, 2021, nature Communications, 12:4242. [Online]. Available: https://doi.org/10.1038/s41467-021-24496-9

[9] L. Organick, S. D. Ang, Y.-J. Chen, R. Lopez, S. Yekhanin, K. Makarychev, M. Z. Racz, G. Kamath, P. Gopalan, B. Nguyen, C. N. Takahashi, S. Newman, H.-Y. Parker, C. Rashtchian, K. Stewart, G. Gupta, R. Carlson, J. Mulligan, D. Carmean, G. Seelig, L. Ceze, and K. Strauss, "Random access in large-scale DNA data storage," *Nature Biotechnology*, vol. 36, no. 3, pp. 242–248, 2018, pMID: 29457795. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/29457795/

[10] S. M. H. Tabatabaei Yazdi, Y. Yuan, J. Ma, H. Zhao, and O. Milenkovic, "A rewritable, random-access DNA-based storage system," *Scientific Reports*, vol. 5, p. 14138, 2015, received 05 May 2015; Accepted 17 August 2015; Published 18 September 2015. [Online]. Available: https://www.nature.com/articles/srep14138

[11] W. H. Press, J. A. Hawkins, S. K. J. Jones, J. M. Schaub, and I. J. Finkelstein, "HEDGES error-correcting code for DNA storage corrects indels and allows sequence constraints," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 117, no. 31, pp. 18 489–18 496, 2020, pMID: 32675237; PMCID: PMC7414044. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/32675237/

[12] C. Xu, C. Zhao, B. Ma, and H. Liu, "Uncertainties in synthetic DNA-based data storage," *Nucleic Acids Research*, vol. 49, no. 10, pp. 5451–5469, 2021, pMID: 33836076; PMCID: PMC8191772. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/33836076/

[13] H. Yeom, N. Kim, A. C. Lee, J. Kim, H. Kim, H. Choi, S. W. Song, S. Kwon, and Y. Choi, "Highly accurate sequence- and position-independent error profiling of DNA synthesis and sequencing," *ACS Synthetic Biology*, vol. 12, no. 12, pp. 3567–3577, 2023, pMID: 37961855; PMCID: PMC10729760. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/37961855/

[14] R. N. Grass, R. Heckel, M. Puddu, D. Paunescu, and W. J. Stark, "Robust chemical preservation of digital information on DNA in silica with error-correcting codes,"

Angewandte Chemie International Edition, vol. 54, no. 8, pp. 2552–2555, 2015, pMID: 25650567. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/25650567/

[15] Y. Erlich and D. Zielinski, "DNA Fountain enables a robust and efficient storage architecture," *Science*, vol. 355, no. 6328, pp. 950–954, 2017, pMID: 28254941. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/28254941/

[16] Z. Ping, S. Chen, G. Zhou, X. Huang, S. J. Zhu, H. Zhang, H. H. Lee, Z. Lan, J. Cui, T. Chen, W. Zhang, H. Yang, X. Xu, G. M. Church, and Y. Shen, "Towards practical and robust DNA-based data archiving using the yin–yang codec system," *Nature Computational Science*, vol. 2, no. 4, pp. 234–242, 2022, pMID: 38177542; PMCID: PMC10766522. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/38177542/

[17] A. Isidro-Llobet, M. N. Kenworthy, S. Mukherjee, M. E. Kopach, K. Wegner, F. Gallou, A. G. Smith, and F. Roschangar, "Sustainability challenges in peptide synthesis and purification: From R&D to production," *The Journal of Organic Chemistry*, vol. 84, no. 8, pp. 4615–4628, 2019, pMID: 30900880. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/30900880/

[18] A. N. Gutman *et al.*, "Predicting the success of Fmoc-based peptide synthesis," *ACS Omega*, vol. 7, no. 29, pp. 25 805–25 814, 2022, pMID: 35847273; PMCID: PMC9280948. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/35847273/

[19] T. Muth and B. Y. Renard, "Evaluating de novo sequencing in proteomics: already an accurate alternative to database-driven peptide identification?" *Briefings in Bioinformatics*, vol. 19, no. 5, pp. 954–970, 2018. [Online]. Available: https://doi.org/10.1093/bib/bbx033

[20] C. N. Poston, R. E. Higgs, J. You, V. Gelfanova, J. E. Hale, M. D. Knierman, R. Siegel, and J. A. Gutierrez, "A quantitative tool to distinguish isobaric leucine and isoleucine residues for mass spectrometry-based de novo monoclonal antibody sequencing," *Journal of the American Society for Mass Spectrometry*, vol. 25, no. 7, pp. 1228–1236, 2014, pMID: 24845350. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/24845350/

[21] Y. Xiao, M. M. Vecchi, and D. Wen, "Distinguishing between leucine and isoleucine by integrated LC-MS analysis using an orbitrap fusion mass spectrometer," *Analytical Chemistry*, vol. 88, no. 21, pp. 10 757–10 766, 2016, pMID: 27704771. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/27704771/

[22] S. L. Rössler, N. M. Grob, S. L. Buchwald, and B. L. Pentelute, "Abiotic peptides as carriers of information for the encoding of small-molecule library synthesis," *Science*, vol. 379, no. 6635, pp. 939–945, 2023, pMID: 36862767; PMCID: PMC10064805. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/36862767/

[23] B. Luo, S. Gao, M. Wu, X. Dong, X. Deng, and H. Hu, "Peptide-encapsulated hydrogels for long-term data preservation," *Communications Materials*, vol. 6, p. 183, 2025. [Online]. Available: https://doi.org/10.1038/s43246-025-00915-y

[24] A. Zhang, L. Wang, X. Zhai, Y. Xiao, Y. Wu, Y. Zhao, K. Liu, J.-S. Zheng, and D. Chen, "Composite mapping for peptide-based data storage with higher coding density and fewer synthesis cycles," *Advanced Science*, vol. 12, no. 27, p. e2503790, 2025, pMID: 40285644; PMCID: PMC12279161. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/40285644/