Abstract of "Secure Data Compression and Error Correcting Codes for Networks and Cloud Storage" by James Alan Kelley, Ph.D., Brown University, May 2015.

We present several novel constructions—combining cryptography, error correcting codes (ECCs), and data compression—that find ready application in enhancing security and fault-tolerance in cloud storage. We demonstrate this by presenting a simple (yet novel) secure cloud storage scheme (which can be used on top of any cloud service provider) that provides strong guarantees of integrity and fault-tolerance, and we show the different enhancements possible using our constructions.

Our constructions provably achieve strong theoretical properties and are quite practical as well. First, we consider the problem of combining data compression with encryption to provide a primitive that performs both operations at once. This work provides the first formal definitions of security for schemes that combine compression and encryption. We present two compressing ciphers that are the first to provably achieve these strong guarantees of privacy and security. Moreover, one construction is quite practical and provides data compression ratios and speeds comparable to standard algorithms, as demonstrated with a detailed set of experiments.

Second, we utilize cryptographic primitives to enhance erasure codes to withstand adversarial corruption of the encoded data. As part of this, we present a new adversarial model for ECCs which is more powerful than previously considered. We then provide two constructions, called *authenticated error correcting codes*, that transform an erasure code into an ECC and are provably secure in our model. The first scheme combines digital signatures and list decoding while the second uses a message authentication code (MAC), a non-malleable cipher, and a pseudorandom permutation.

Finally, we present cryptographically enhanced LT codes (a fast, rateless *erasure* code) which are able to provide *error* correction over an adversarial channel. LT codes encode data via a sparse bipartite graph where each output symbol is the XOR of a random subset of the message symbols. All prior work on LT codes only considered random erasures or random errors (e.g., additive white Gaussian noise) and would fail under adversaries that exploit the encoding graph. We define a new framework for analyzing the security of rateless codes and provide three provably secure constructions: (1) a basic scheme that is used as a subroutine in the other schemes; (2) a scalable, block-oriented fixed-rate scheme; and (3) a scalable, block-oriented *rateless* scheme. All of these schemes maintain both asymptotic and practical efficiency—the latter we demonstrate experimentally.

Secure Data Compression and Error Correcting Codes for Networks and Cloud Storage

by

James Alan Kelley

B.A., Boston University; Boston, MA, 2009

Sc.M., Brown University; Providence, RI, 2011

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2015

This dissertation by James Alan Kelley is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____

Roberto Tamassia, Director

Recommended to the Graduate Council

Date _____ _____

Rodrigo Fonseca, Reader

Date _____ _____

Nikos Triandopoulos, Reader

Approved by the Graduate Council

Date _____ _____

Peter M. Weber
Dean of the Graduate School

Vita

James Kelley was born in sunny San Bernardino, California and soon thereafter was whisked away to just outside Redmond, Washington where he grew up and developed a sophisticated appreciation for clouds. As a child 6 years of age, he fell in love with mathematics, relishing every problem set. This passion accompanied him all through school into junior high when he discovered the fascinating world of computers. Little by little, he learned how these intricate machines worked. Eventually, he came across the field of cryptography: a fusion of his two passions. Initially captivated by tales of spies and mathematical trickery, he began to appreciate the deep challenges in cryptography and, more generally, those inherent to computer security. It was at this time in high school that he knew that tackling the challenges of computer security would be endlessly rewarding, and he could not imagine doing anything else.

In 2005, he attended Boston University and double majored in mathematics and computer science with a focus on cryptography, computer systems, and security. He graduated *summa cum laude* in 2009, received the College Prize for Excellence in Computer Science along with the Robert E. Bruce Memorial Prize in Mathematics, and was inducted into *Phi Beta Kappa* honor society. He then enrolled in the computer science doctoral program at Brown University under the tutelage of Roberto Tamassia: the ever patient and accommodating. While attending Brown, he married his wife Tatyana, and they had two children, Alexei and Ksenia, who all have been continual lights in his life.

Acknowledgments

There are many without whom this thesis would not have been finished, and I would not be here. First, I would like to thank our Lord God for the gift of this time at Brown and the growth and maturation that time has given me. The love, support, and encouragement of my wife Tatyana have been crucial in keeping my efforts focused and my sanity preserved. Our two children, Alexei and Ksenia, are continual sources of levity and joy and provide a welcome change of pace to the grinding edit-compile-debug cycle.

My parents and family provided a solid, loving foundation on which I could easily pursue my studies. Along the way, I have been fortunate to have good friends to help keep me grounded, including Sam, Ivan, Greg, The Josh, and James. All of the parishioners at Holy Resurrection Orthodox Church have been a second family to me, especially Razvan and Oana Veliche. Fr. Patrick Tishel and Fr. Michael Kon have both been full of gentle guidance and sage advice while navigating the many roads of life. Were it not for these friends, family, and pastors, I would be a different person.

Since coming to Brown, Roberto has been a gentle, flexible, and patient advisor whose wisdom, guidance, and feedback have proven invaluable. I have also been fortunate to collaborate with both Nikos Triandopoulos and Ari Juels, whose insights and observations have been crucial in rigorizing and smoothing out many of the rough edges in early versions of this work. Rodrigo Fonseca has repeatedly provided helpful feedback on this work and quite often gave clever tips and tricks for improving implementations of the schemes in this thesis.

The Brown Computer Science department would grind to a halt were it not for the tireless efforts of the administrative and technical staff. Lauren Clarke has been a constant friendly face as well as exceedingly responsive and helpful in managing the bureaucracy of graduate life. My MaxBuilt machine provided by the technical staff has served me well these past years. I would also like to thank John Bazik for his tender love and care given to the compute grid, which was used extensively

in benchmarking Falcon codes. The work on the squeeze ciphers benefited greatly from the feedback of James Lentini, Peter Shah, and Shiva Chaitanya of NetApp during an internship in the summer of 2013. Lastly, I would like to thank my undergraduate professors Gene Itkis, Leo Reyzin, and Azer Bestavros for their guidance and encouragement to pursue graduate studies.

CONTENTS

⋆ Parts of this dissertation have been published previously. In particular, Chapter 2 is an extended version of [79] and Chapter 4 is an extended version of [73].

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER **One**

Securing Cloud Storage

1.1 Introduction

The cloud, an amorphous collection of computing resources, provides several benefits that can enhance (and sometimes replace) traditional information technology (IT) services, and is quickly becoming an integral part of IT operations around the world. At a basic level, the cloud provides an outsourced IT infrastructure that is best thought of as "utility computing" where a subscriber pays for exactly what they use (like electricity). Different cloud providers offer different usage models, including: Infrastructure-as-a-Service, where users are allotted a certain amount of hardware resources to use as desired (e.g., Amazon EC2 [2]); Platform-as-a-Service, where the hardware and a software stack is provided and users can easily build their own applications on top of it (e.g., Google App Engine [52]); and Software-as-a-Service, where the hardware and software are setup by the provider and the user simply pays to use the application (e.g., Drupal Gardens [44]). This outsourcing allows companies to lower their IT costs by having a cloud provider manage the actual hardware/platform/software in their (remote) data centers. Also, using the cloud can enable better disaster recovery since it readily provides strong fault-tolerance through geographic distribution and replication of both data and computation. While it seems unlikely that the cloud will completely dislodge all business IT (e.g., due to regulatory restrictions), the cloud is here for the foreseeable future and its use will continue to grow.

When considering whether or not to invest in adopting the cloud (even partially), businesses are keen to understand the risks involved, especially the security risks. By outsourcing their infrastructure

to a third-party, businesses can become more vulnerable to various threats, both accidental and malicious. Cloud services regularly experience downtime due to software bugs [125, 159], problems with upgrading the cloud infrastructure [33, 160], and even weather [113, 161]. Indeed, the bug-induced outage described in [159] resulted in data loss for a portion of Amazon's customers (specifically, those who used the Elastic Block Storage service). Mistakes can also undermine the security of the cloud, such as in 2011 when a bug in the authentication mechanism of Dropbox—a cloud storage provider—allowed anyone to access any account using any password [37]. Malicious attacks on cloud providers have also occurred. For instance, in 2012, an employee at Dropbox had the credentials for their own Dropbox account stolen, resulting in the theft of an internal company document containing Dropbox user email addresses [38]. Even cloud giants that have world-class security teams, such as Google, can fall victim to hackers.[1]

In general, cloud providers give few guarantees about anything. For instance, the Amazon S3 object storage service guarantees at least 99.9% uptime during any monthly billing cycle (see [5]). Note that the guarantee is one solely of *availability* and no guarantees are made about data integrity or consistency (though, they do provide strong guarantees of *durability*). For Amazon's "elastic compute" EC2 service, they guarantee 99.95% uptime where downtime is defined exclusively as "region unavailability" where "*all* of your running instances have no external connectivity" (emphasis added, see [4]). This definition purposefully excludes any downtime caused by, say, a single rack of servers going down and (possibly) taking down the majority (but not all) of your running instances. As long as a few of your compute instances are running and have external connectivity, it does not count as downtime.

Since so few guarantees are provided, particularly in the cloud storage realm, many researchers and technologists have been working to layer additional services on top of cloud storage to improve security, integrity, and data consistency guarantees. Several authors have sought to protect against data corruption from byzantine failures in cloud servers, e.g., [17, 24, 51, 62, 130]. Other authors have sought to add strong consistency guarantees and the ability to detect violations of cloud service level agreements (see [130, 149]). In this thesis, we provide several tools that can readily enhance the efficiency and adversarial-resilience of many (secure) cloud storage schemes, as demonstrated by enhancing a simple, yet novel, secure cloud storage scheme, described next. The individual tools are

[1] In late 2009, Google was hacked by unknown persons, in a campaign dubbed Operation Aurora [54], that stole intellectual property from Google and possibly sought information on dissidents in China. The attack was on Google itself and not their cloud infrastructure specifically.

fully detailed in subsequent chapters, and we cover the previous and related work for these schemes in their respective chapters. We do not address the issue of data consistency models in this work.

Contributions and Organization. In this thesis, we present several general-purpose tools that find ready application in enhancing the privacy, data integrity, and fault-tolerance of cloud storage. We demonstrate this applicability by using these tools to enhance a simple, yet itself novel, architecture for increasing the resilience of cloud storage to data loss and corruption (without modifying the service provider) which we detail next.

The remainder of this thesis is organized as follows. In the rest of this chapter, we provide a high-level description of our simple architecture, with intuitive definitions for some of the basic tools used (presented as needed), and detail how our constructions can strengthen and improve the efficiency of the scheme.

Chapter 2 describes our work on the Slow and Fast Squeeze ciphers, two new ciphers that provide data compression while also being provably secure—the first such schemes.[2] The algorithms are derived from the well-known LZW compression algorithm [165] that allows for efficient implementations, which we demonstrate through a thorough experimental evaluation. We also provide a new definitional framework for proving the security of combined compression-encryption schemes and show that it relates in a simple way to the standard definitions of security for ciphers. Moreover, we prove that our constructions achieve the strongest possible security in this model. This work was done in collaboration with James Lentini, Peter Shah, and Shiva Chaitanya of NetApp's Advanced Technology Group.

Chapter 3 presents two general constructions that turn an *erasure* code into an *error correcting* code—with only a small decrease in the information rate of the code—such that they can withstand (computationally-bounded) adversarial corruption of the data. Indeed, we define a new adversarial model for analyzing the error correcting properties of certain codes when attacked by computationally-bounded adversaries. This model is more powerful than previously considered and is able to capture more real-world adversarial behaviors. Our schemes are provably secure in this model and, moreover, achieve better error correction in this model than previous work. Our first construction combines list decoding with collision-resistant hash functions and digital signatures to correct errors more efficiently

[2] Previous work was *not* provably secure, and, indeed, was often quite insecure, see Chapter 2, Section 2.8.

than previous list-decoding-based schemes (i.e., [111]). The second construction utilizes a non-malleable cipher, a message authentication code, and a generic pseudorandom permutation to achieve the same error correction as our first scheme without using list decoding. We call these *authenticated error correcting codes*. This work was performed jointly with Nikos Triandopoulos of RSA Laboratories.

Chapter 4 introduces a novel family of error correcting codes based on LT codes [103], an efficient family of rateless erasure codes. In particular, we enhance LT codes with a combination of a pseudorandom generator, semantically secure encryption, and a message authentication code to form an error correcting code able to withstand *adversarial* data corruption.[3] We call this family of codes *Falcon codes*. These codes are a generic construction that can be used as a drop-in replacement for (almost) any LT code[4] and provide immediate tolerance to malicious corruption of the encoded data—e.g., using Falcon codes in the LT-coding step of Raptor and Online codes (which are erasure codes) enables the codes to withstand adversarial corruption. As part of this work, we have developed a new adversarial model for analyzing the security of rateless codes—since previous models were only applicable to fixed-rate codes—and prove the security of our schemes in this model. Additionally, we provide two *scalable* variants that can encode and decode large files more efficiently than the basic scheme. We experimentally evaluate all of our schemes and demonstrate their practicality. This work was performed jointly with Ari Juels of Cornell Tech and Nikos Triandopoulos of RSA Laboratories.

Finally, we conclude in Chapter 5 with a summary of this thesis.

1.2 High-level Architecture

In this section we first outline several properties desired for cloud storage and then outline our architecture that achieves these properties. Then, we detail how the tools presented in this thesis can enhance different aspects of this architecture (including efficiency, integrity, and privacy), and we outline some of the trade-offs involved.

1.2.1 Desired Properties of Cloud Storage

There are numerous desired features for cloud storage including data availability, integrity, and privacy as well as overall system efficiency. Since cloud storage moves data outside of the data owner's

[3] Previous work considered only *random* errors, such as additive Gaussian white noise.

[4] Our construction places lower-bounds on the symbol size of the code to achieve strong guarantees of security—e.g., the symbol must be at least 128-bits in size—and so it cannot replace an arbitrary LT code.

network perimeter, the owner has less control over the management of the data, but data owners can preprocess their data to achieve the desired levels of reliability, integrity, and privacy. For instance, error correcting codes[5] can readily increase the integrity and availability of the data while a simple application of encryption can ensure privacy. If data corruption is detected, the system must also be able to recover efficiently and repair the data, as it is inefficient to re-upload an entire file if only a small portion was damaged. Another desirable feature for cloud storage is *public verifiability* of the integrity of the data. That is, while an organization may outsource (some of) their data to the cloud, it may not have the resources to continually check the integrity and availability of their data. Public verifiability allows a (not necessarily trusted) third-party auditor to examine the data stored in the cloud for any problems. This can increase the security of the data while not placing a large burden on the organization using the cloud. Moreover, public verifiability helps resolve any service disputes between the user and the cloud provider.

Additionally, since a great deal of data is *dynamic*, an organization must be able to update the data in the cloud without paying large costs, both in bandwidth and computation. The cloud loses much of its utility if, whenever a portion of a file is accessed, all (or even most) of the hosting servers must be contacted and/or a majority of the file must be downloaded. Finally, the bulk upload/download costs must be reasonable; otherwise, the setup cost of switching to the cloud will be too great. To summarize, the following are desirable properties for a cloud storage system:

1. Tolerance of partial (adversarial) data loss/corruption;
2. Efficient repair of data;
3. Public verifiability;
4. Minimal overhead for small reads/writes (should be proportional to the operation size); and
5. Minimal storage/computational/communication overhead for getting or putting a file;

The first two items can be achieved by using error correcting codes on the data and utilizing striping of the ECC (striping is defined later). The third property can be achieved by simply signing each piece of data with an asymmetric key pair and giving the public key to an auditor. The last two properties, though, require some care to achieve, and this is where we find practical applications for the constructions detailed in this work.

[5] An error correcting code is a message encoding scheme that can tolerate some amount of corruption of the encoding and still decode back to the original message. Codes with fixed input and output lengths (denoted k and n respectively) are called *fixed-rate* codes with the ratio k/n called the *rate* of the code and can recover the input message with any k out of n of the code symbols.

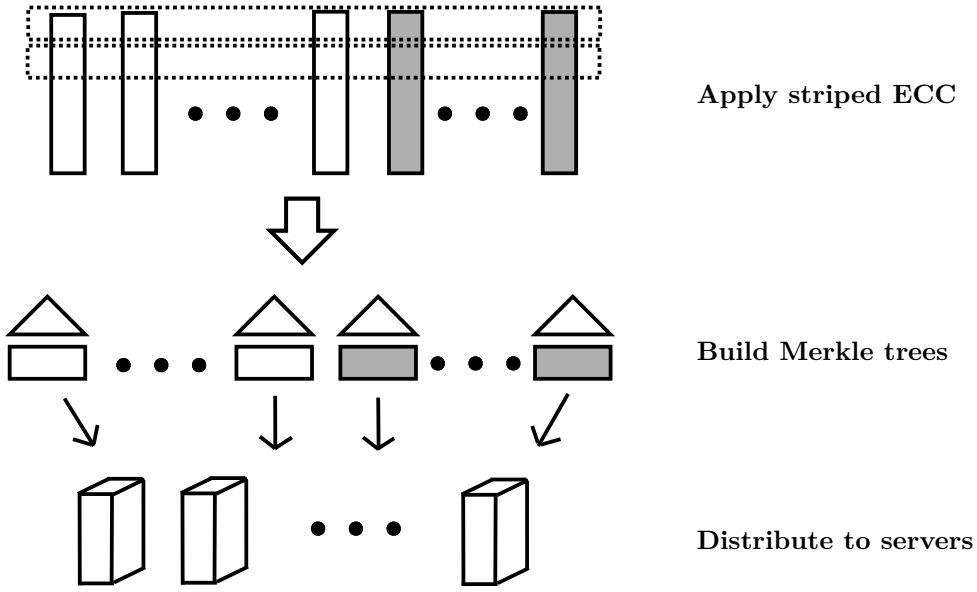


Figure 1.1: The basic secure storage scheme. The dashed boxes indicate a stripe of the ECC, shaded boxes are the parity symbols, and the triangles are the Merkle trees.

1.2.2 A Simple Scheme

Suppose we have a file F we would like to store on n servers in the cloud. In our scheme, we simply apply a systematic error correcting code[6] (e.g., a systematic Reed-Solomon code) to F. Specifically, F is broken up into k pieces of length l bytes, called *message symbols*, and then encoded into n *code symbols* (also l bytes in length). The code symbols are then divided into blocks of b bytes and a Merkle tree is built over each symbol with the blocks forming the leaves of the tree.[7] The Merkle tree allows us to efficiently (and with overwhelming probability) detect any corruptions of the data and discard the spurious data, allowing us to utilize an *erasure code*[8] for the encoding. This reduction of corruptions to erasures via cryptographic means is not novel and, indeed, has been used extensively (see, for example, [20, 24, 63, 88]). But, building a (distinct) Merkle tree over each symbol is a novel twist on typical usage of Merkle trees in authenticating data: e.g., distillation codes [76] have a single

[6] An error correcting code is *systematic* if the original data forms part of the encoded output. Systematic codes enable much more efficient decoding in the common case where only a small portion (or none) of the data is corrupted.

[7] A Merkle tree is a binary hash tree used to authenticate a set of elements using just a collision-resistant hash function. Each internal node of the tree contains the hash of the contents of its two children and a leaf contains the hash of the corresponding set element. The hash value in the root node is published publicly. An element is authenticated by sending the element and the hashes of the sibling nodes on the path from the element to the root. The verifier recomputes the hashes on the path using the element and the sibling hashes and compares the final result to the published root hash: accepting if they are equal and rejecting if not.

[8] An erasure code is an ECC that can correct data erasures (i.e., lost data) but not data errors. Erasure codes are typically much more efficient that error correcting codes.

tree for the encoded message and each *entire symbol* forms a leaf of the tree. Other works that build a single Merkle tree over the entire encoding include [24, 63].

We call the "plain data" code symbols the *systematic* symbols and all other symbols the *parity* symbols. With a Merkle tree for each code symbol, the client stores the root hash of each Merkle tree and then sends the encoded data with the trees to the n servers. If we use an $[n, k]$ Reed-Solomon code[9] (i.e., where at least k out of n uncorrupted code symbols are needed to decode without errors), then the client stores n hashes for F. This cost is reasonable since the client must store the information identifying the servers hosting the data anyway, and the hashes could easily be included with the server IDs.

Since each symbol is l bytes and divided into blocks b bytes in size, there are $m = l/b$ blocks per symbol. To support small reads, the client requests the desired block, and the server returns the hashes of the siblings on the path to the root of that symbol's Merkle tree. The client can then verify the integrity of the block in $O(b + \log m)$ time by recomputing the appropriate hashes (assuming hash outputs are of constant size, hashing a hash takes $O(1)$ time). The communication overhead is also $O(b + \log m)$.

Code Striping. To support efficient writes, we cannot use a Reed-Solomon code as-is. As currently described, our system encodes the file using symbols l bytes in size, with l potentially being quite large. Any change to a symbol could result in all $n - k$ parity symbols being completely different, i.e., a small change results in updating $(n - k + 1)l$ bytes. To avoid this, we use a technique called *code striping*. Striping is a technique where the input message is broken up into small segments which are then independently encoded using the same error correcting code (i.e., the same n and k) but over a smaller finite field. The resulting code symbols are then interleaved to produce the final larger code symbols: in particular, all of the i-th code symbols (from each stripe) are concatenated together to form the i-th aggregate code symbol.[10] Striping allows for fine-grained updates to the data since any change is localized to the stripes containing the change. For example, looking at Figure 1.1, if the top-most block in the left-most symbol is updated, then the *only* blocks that must be updated are those contained within the same dashed-box (which indicates a single stripe of the ECC).

[9] An $[n, k]$ Reed-Solomon code breaks a message up into k pieces of l bytes each, and the interprets them as coefficients of a polynomial $p \in \mathbb{F}[x]$, where $\mathbb{F} = GF(2^{8l})$. The encoding is then the evaluation of p at n distinct points.

[10] Striping an ECC is an extension of data striping found in RAID systems (hard disk arrays where data is spread across multiple disks for greater redundancy and throughput [124]). In these systems, consecutive data bytes (or blocks) are written to the disks in a round-robin fashion. For RAID systems that apply (systematic) error correcting codes to the data, the i-th disk in the array corresponds to the i-th aggregate code symbol for the data.

Code striping was developed to make the encoded data more robust to burst errors while in transmission, as a burst error in a single large symbol would "spread out" the error over many smaller encoded segments. Striping also has the benefit that, when encoding large inputs, it can be much faster than the "plain" (non-striped) code. For example, using a Reed-Solomon code, if we break the file up into l-byte symbols, the arithmetic operations for the code are performed over $GF(2^{8l})$ while striping can use a field as small as $GF(2^8)$, which allows for efficient field operations.[11]

An important detail for striping to be efficient in this scheme is to compose a stripe out of symbols that are a distance of l bytes apart (i.e., the size of one aggregated code symbol), instead of simply encoding consecutive bytes. This allows the large code symbols to be formed without actually shuffling any data around (at the cost of losing some spatial locality when encoding/decoding data far apart in the file), and it simplifies reading the data since consecutive bytes in the input file will still be consecutive in the (systematic) encoding of the file.

Updates. To perform updates on the encoded data, we can exploit the algebraic properties of the Reed-Solomon code to minimize the amount of computation. A Reed-Solomon code works by parsing the input file F into a sequence of k elements in a finite field $\mathbb{F}$ (called *symbols*), which are interpreted as coefficients of a degree $k-1$ polynomial $p(x) \in \mathbb{F}[x]$ (we assume that the coefficient of the highest-order term is non-zero). The output code symbols are simply the evaluation of $p(x)$ on a set of n distinct points (denote these by $\alpha_1, \ldots, \alpha_n$). For example, if we want to change symbol s_i to s'_i, creating the new polynomial $p'(x)$. The difference between $p'(x)$ and $p(x)$ as polynomials is simply $(s'_i - s_i)x^i$; moreover, $p'(\alpha_j) - p(\alpha_j) = (s'_i - s_i)\alpha_j^i$ for any $\alpha_j \in \{\alpha_1, \ldots, \alpha_n\}$. Thus, to generate the "patches" necessary to update a single encoded block, we need to perform at most a single subtraction and n multiplications in the finite field (since α_j^i can be precomputed). If we use a systematic code, then we only need to perform $n-k+1$ multiplications as the $k-1$ remaining (systematic) symbols do not need to be updated. These updates must be applied to the corresponding block in each parity symbol, requiring the block be read (for each symbol) and a single addition performed. Thus, we require $n-k+1$ multiplications and $n-k+1$ additions to generate and apply the patches. The communication overhead is $O((n-k+1)(b + h \log m))$ since each block must be read and then written back, including the Merkle tree updates. There is also the additional overhead of $O((n-k+1)\log m)$ hash operations needed to verify the data and parity symbols.

[11] For example, multiplication in $GF(2^8)$ can be implemented via a lookup table of size $(2^8)^2/2 = 2^{15}$ bytes.

Summary of Efficiency. The scheme given in this section has several desirable properties. First, it can tolerate the loss and/or corruption of at most $n-k$ symbols, the maximum for an $[n,k]$ code. Second, the scheme is publicly verifiable since the root hashes of the Merkle trees can be given to the auditor and verified without exchanging any secrets. Third, the storage overhead is $(n-k)(|F|/k)+n(2m-1)h = (\frac{1}{R}-1)|F|+n(2m-1)h$, where $m=l/b$ is the number of b-byte blocks per l-byte symbol, h is the size of a hash in bytes, and $R=k/n$ is the rate of the code. We observe that if $|F|$ is large and $b \gg h$, then the $n(2m-1)h$ term is small. This combination of code striping and per-symbol Merkle trees allows for much finer-grained reads and writes to the data than previous schemes. Finally, the client must store only nh bytes for the root hashes of the Merkle trees. We summarize the costs of the scheme in Table 1.1 and detail them in the following list.

- The overhead for putting a file is $O((n-k)k(|F|/(kt))+nm) = O((n-k)(|F|/t)+nm)$, where t is the size of a symbol used in striping. Evaluating a degree $k-1$ polynomial at a point can be performed in $O(k)$ time with k multiplications and k additions using Horner's method. We assume that multiplications and additions in a finite field take constant time.[12] Since the code is systematic, we only need to perform $n-k$ polynomial evaluations to generate the parity symbols, and this must be done $|F|/(kt)$ times. Each symbol also requires $2m-1$ hash operations to compute the Merkle tree on each (aggregate) symbol.
- Fetching an entire file requires $O(|F|+(\frac{1}{R}-1)|F|+hnm) = O(\frac{1}{R}|F|+hnm)$ communication and potentially $|F|/(tk)$ decodings, each of which takes $O(k^2)$ time, in addition to verifying the symbols via the n Merkle trees.[13]
- Reading a single block requires $O(b+h\log m)$ communication and $O(b+\log m)$ computation.
- Writing a single block requires $n-k+1$ finite field multiplications and $n-k+1$ additions. There are also $O((n-k+1)\log m)$ hashing operations and a communication cost of $O((n-k+1)(b+h\log m))$.
- Recovery of a corrupted block takes n reads, b/t stripe decodings, and at most $n-k$ writes (since the code cannot tolerate more corruption than that), for a computational complexity of $O(n(b+\log m)+(b/t)k^2+(n-k)\log m)$.

[12] This is a reasonable assumption when using striping since the field used is typically $GF(2^8)$ or $GF(2^{16})$ and multiplication tables can be precomputed. For large symbols in $GF(2^b)$, each multiplication can be computed in $O(b\log b\log\log b)$ using the discrete Fast Fourier Transform.

[13] Reed-Solomon decoding can be done via the inverse discrete Fourier Transform in $O(n\log n\log\log n)$ time, but this algorithm is impractical for small values of n and k. The Berlekamp-Massey algorithm can decode Reed-Solomon codes in $O(k^2)$ time [107].

Table 1.1: The efficiency of the simple secure storage scheme (without our enhancements) for various operations. Communication cost is the number of bytes transmitted.

	Computation cost	Communication cost
Bulk put	$O((n-k)(\|F\|/t) + nm)$	$O(\frac{1}{R}\|F\| + hnm)$
Bulk get	$O(k\|F\|/t + nm)$	$O(\frac{1}{R}\|F\| + hnm)$
Small read	$O(b + \log m)$	$O(b + h \log m)$
Small write	$O((n-k+1)\log m)$	$O((n-k+1)(b + h \log m))$
Repair	$O(n(b + \log m) + (b/t)k^2 + (n-k)\log m)$	$O(n(b + h \log m) + (n-k)h \log m)$

Small Optimizations. A simple optimization that we can perform is to replace the Merkle tree with a multi-way authenticated skiplist [157] (which is a generalization of an authenticated skiplist [50]). A multi-way skiplist, in expectation, performs the optimal number of hashing operations, and in [157] it is shown experimentally that the expected number of hashes stored is $\approx 1.41m$, which is less than the $2m - 1$ required for a Merkle tree.[14] In addition, an authenticated skiplist is more flexible than a Merkle tree as it more readily supports data insertion and deletion while preserving its $O(\log n)$ access guarantees (with high probability).

A second optimization for our system is aimed at small files (e.g., a few hundred bytes). For a small file, it is easy to avoid the complexity and cost of Merkle trees altogether and simply sign and encode the data. The client would then fetch the entire file for any reads or writes—which could be done in a single network round-trip. For files that are slightly larger (e.g., few kilobytes), we can to store the hash of each *entire symbol* instead of the root of a Merkle tree. We would have the same data corruption resilience guarantees while simplifying the protocols and algorithms.

1.2.3 Avoid Storing Root Hashes

The above scheme works fairly well, but it requires $O(n)$ storage at the client (where again n is the number of servers). We can reduce this overhead to $O(1)$ by having the client possess an asymmetric key pair (pk, sk) and use it to sign the concatenation of the hashes before replicating the hashes and signature to each server. This increases the per-server storage overhead by $nh + s$ bytes, where s is the signature length and h is the hash length. We call the hashes and signature the *authentication information* for the encoded file.

To reduce this overhead at the servers, instead of using n-fold replication, we can simply use an error correcting code to redundantly store them, giving an overhead of $\frac{1}{k}(nh + s)$ per server instead

[14] Note, the precise expected number of hashes stored depends on the parameterization of the skiplist. Here, we assume that $p = 1/2$, where p is the probability that an element in layer i appears in layer $i + 1$ of the skiplist.

of $nh + s$. Using a systematic code, in the best case the client will need to communicate with k servers (in parallel) with a total communication cost of $nh + s$ (which is identical to the best case of replication), and at worst n servers with a total communication cost of $\frac{n}{k}(nh + s)$ (instead of $(n - k + 1)(nh + s)$ for the replication scheme).

However, the data corruption tolerance in this setup decreases from $(n - k)$ to $(n - k)/2$ (which is the error correcting capacity of Reed-Solomon codes), since there is no authentication information for the encoded hashes and signature. That is, there is no mechanism to detect corruption in the encoded authentication information, and so we cannot detect and discard corrupt symbols. Thus, the corruption tolerance gained by using Merkle trees is lost. To counteract this, we leverage the technique of *list decoding* where, instead of outputting a single value, the decoder for the ECC is permitted to output a list of values (instead of a single value), with the correct value guaranteed to be in the list. This relaxation allows us to correct up to $n-k$ errors for a Reed-Solomon code.[15] To locate the correct value in the list, we utilize the signature on the hashes, accepting the codeword whose signature verifies. Against a computationally-bounded adversary, with overwhelming probability, only the correct data will have a valid signature (first proven in [106] and rediscovered in [111]). That is, signing the input message allows us to disambiguate the list decoding and achieve *unique* decoding. Note that no special encoding must be done to perform list decoding; the list decoding algorithm works as-is with a standard Reed-Solomon code. Also, as before, we can support updates to the authentication information (i.e., the hashes and the signatures) using the algebraic properties of Reed-Solomon codes.

Using list decoding on the encoded file itself is possible, but quite expensive since the code symbols could be rather large (this usage is the scheme described in [111]), and the algebraic operations on these large symbols have non-trivial costs. The construction presented here uses list decoding on much *smaller* code symbols—since the input message was just a few hashes and a signature—so the list decoding algorithms can operate more efficiently. That is, this hash-sign-encode combination (for the authentication information) provides an efficient, generic construction for transforming an *erasure* code applied to an input message into an *error-correcting* code for that message (over a computationally-bounded channel). This construction we call an *authenticated error correcting code*, and we will analyze it in detail in Chapter 3 along with an alternative construction that does not require list decoding.

[15] See [58] for a description of a list decoder for Reed-Solomon codes that can correct up to $n - k$ errors.

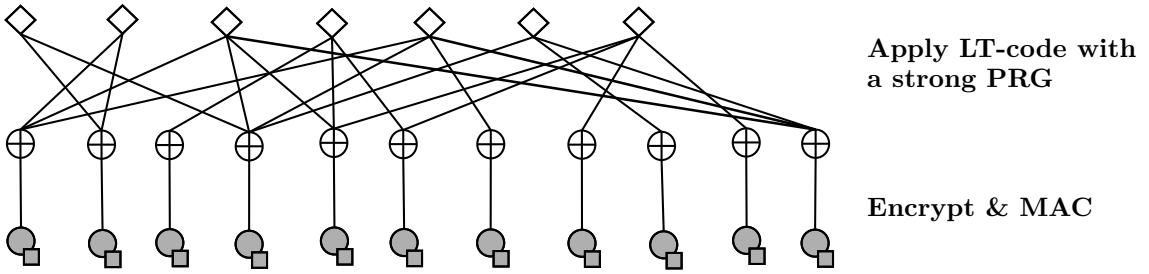


Figure 1.2: The authenticated LT code.

Also of note is that applying a list-decodable ECC to the authentication information completely separates the corruption of the data from the corruption of the hashes (i.e., they can be corrupted independently). This enables the distribution of the authentication information to a separate set of n servers that, themselves, can be corrupted independently of the n data servers. This feature is not possible with previous constructions combining cryptography and ECCs.

1.2.4 Faster Encoding and Decoding

Encoding with an $[n, k]$ Reed-Solomon code asymptotically takes $O(n \log n \log \log n)$ time using the Discrete Fast Fourier Transform, but for practical (i.e., small) values of n and k Horner's method is the most efficient, taking $O(nk)$ time to evaluate a degree $k-1$ polynomial at n points. (Again, this assumes that field operations take a constant amount of time.[16]) The quadratic cost for encoding and decoding is a bottleneck for large files, and thus, it is desirable to seek a faster (both asymptotically and practically) encoding and decoding algorithm that can provide the same error correction guarantees. LT codes [103] are a family efficient erasure codes that provide $O(k \log k)$ time encoding and decoding while ensuring decodability with high probability and being much faster than Reed-Solomon codes. LT codes operate by generating a random, sparse bipartite graph with message symbols in one partition and code symbols in the other. The value of a code symbol is simply the XOR of its neighboring message symbols. The degree of each code symbol is selected according to the *robust soliton distribution* (described in the original LT code paper [103]), and then the neighbors are selected uniformly at random. LT codes can produce a practically *limitless* stream of code symbols and, hence, do not have a fixed rate. Such codes are call *rateless* (or *fountain*) codes.

[16] If we do not use striping and simply have large input symbols, the cost of field operations becomes non-trivial.

Unfortunately, LT codes provide guarantees over *random* channels rather than adversarial ones—e.g., where each symbol is erased with some fixed probability. Moreover, the error correction of these codes is quite limited as the errors are assumed to come from a fixed distribution, e.g., additive Gaussian white noise. In an adversarial setting, errors and erasures can easily become catastrophic.[17] In particular, LT codes allow for decoding to fail with some fixed, user-configurable probability that can be made arbitrarily small, assuming that erasures are random. However, in the presence of adversarial erasures, the probability of decoding failure can be increased well beyond this bound by selectively corrupting or deleting symbols and forcing the received codeword to fall into the (formerly) low probability "bad" case. This attack, which we call *targeted erasure*, was first mentioned in [91] and was called a *distribution attack*; but, the attack was outside of their threat model. The attack was actually implemented by Lopes and Neves in [100] on a code derived from LT codes called Raptor codes[18], and the authors demonstrate that it is possible to increase the failure probability by several *orders of magnitude* from the desired failure probability.

Lopes and Neves also suggest possible mitigations to this attack, but their suggestions are incomplete and do not fully secure the scheme against targeted erasures. In Chapter 4, we introduce *authenticated LT codes*, or *Falcon codes* that provide the same efficiency guarantees as plain LT codes, while also fully securing LT codes against computationally-bounded adversaries. That is, we ensure that any (computationally-bounded) adversary, when corrupting the encoding, is (provably) *no more powerful* than a *random* adversary who simply erases each symbol with some fixed probability. This is the first LT-based code that achieves error (and erasure) correction against adversarial attacks. Additionally, codes derived from LT codes, such as Online and Raptor codes (see [108] and [148], respectively), immediately inherit the security properties of our scheme when using Falcon codes as a drop-in replacement for the LT code. Our construction leverages a combination of a strong pseudorandom number generator (to make the bipartite graph unpredictable), a semantically secure cipher[19] (to hide any information about the graph contained in the symbols), and an unforgeable

[17] While adversarial errors are the worst-case and in some situations may be considered unrealistic, the assumption that errors are random does not hold for, e.g., storage systems. Indeed, errors in storage systems are often coordinated and have a measure of spatial locality, see [9]. Our work protecting against adversarial errors naturally also protects from coordinated storage errors.

[18] Raptor codes [148] are based on LT codes but utilize a "pre-coding" step where a fast erasure code is applied to the input and then the encoded symbols are used as input to the LT encoder. This simple combination allows Raptor codes to achieve high efficiency both practically (being among the fastest erasure codes) and theoretically (with *linear time* encoding and decoding).

[19] A cipher is *semantically secure* if encrypting a message "hides" all information about the message. More formally, any function of the message that can be computed with the ciphertext can also be computed *without* the ciphertext with only a small increase in time to compute the function.

MAC (to detect symbol corruption), shown schematically in Figure 1.2. We note that the use of a semantically secure cipher to protect the bipartite graph gives us privacy for the input data as well.

For the storage scheme, this primitive readily replaces the Reed-Solomon code and gives much more efficient encoding and decoding. Also, to preserve the capability for small reads and writes, we use this primitive in a similar "striping" manner as the Reed-Solomon codes: where input symbols a fixed distance apart are aggregated and become the input for an encoding, and then the output symbols are distributed among servers.

Large Files. Encoding a large file with an LT code can be problematic. For each code symbol, its neighbors are selected uniformly at random from *all possible* input symbols. For very large files—i.e., those that do not fit into memory—this will induce many random read operations from the disk. As an example, say the file is twice as large as the available memory. When producing an output symbol, on average only half of its neighbors will reside in memory. With an expected node degree of $O(\log k)$, this means that there are (on average) $O(\log k)$ disk reads for *every* output symbol. We overcome this bottleneck in Chapter 4 where we provide two scalable variants of our basic Falcon code and experimentally demonstrate their superior efficiency when encoding large files.

A Few Caveats. The first caveat when using Falcon codes is that they are *non-systematic* codes; that is, the output consists only of parity symbols.[20] So for any read operation, the client must download and decode each entire stripe covering the desired data (similarly for writing and repairing).

Secondly, LT-based codes are parameterized by a distribution for the degree of an output node in the bipartite graph. The distribution determines the probability of decoding failure δ and is typically an asymptotically good distribution so that decoding fails with probability at most δ for all *sufficiently large* k. For example, for Raptor codes—which use a variant of the robust soliton distribution—k must be at least 10000 (similar to an LT code).[21] For the secure storage scheme, assuming that the numbers of servers n is fixed and small, this lower-bound on k implies that we must group together many input symbols from a single server and form a larger stripe.

A third caveat is that our constructions require that we use a MAC to authenticate each symbol in the Falcon code, giving $O(1)$ space overhead for each symbol—but, note that MACs have fast

[20] Though the degree distributions are constructed to output many low-degree (e.g., degree 1, 2, or 3) symbols, it is not guaranteed that the first k symbols will be systematic symbols.

[21] Though there are distributions for specific small values of k (e.g., a few hundred), the general design of good distributions for small k is still an open problem.

verification times, often 2–3 orders of magnitude faster than signature schemes.[22] Unfortunately, using MACs causes the scheme to lose public verifiability; but, we can regain it by merely using a public key signature scheme instead. For simplicity in our descriptions we will use a MAC but using it is not intrinsic to the scheme itself.

Finally, an important limitation of this scheme is that is requires the data to be static. The security of Falcon codes depends crucially on the fact that that the adversary does not know the structure of the encoding, i.e., which message symbols have been XORed together to produce a given code symbol. (Otherwise, the adversary can again perform the targeted erasure attack.) Any support for dynamic operations will reveal the structure of the bipartite graph as any update to a single symbol will require updating each parity symbol containing that symbol. Even though the construction is best used for static, archival data, reads and writes of such data are done in bulk and the linear-time encoding and decoding can provide large efficiency gains.

1.2.5 Achieving Privacy and Compression Simultaneously

Two fundamental parts of cloud infrastructure are that: (1) a user pays only for the resources that he uses, and (2) the resources allocated can smoothly scale up as the user's demands increase. This allows users to avoid paying the cost of over-provisioning resources necessary to handle peak load (which may be much larger than the average load). Compressing data before storing it in the cloud decreases the amount of storage that must be provisioned, and hence, lowers the overall costs of the user. Of course, the user must pay the cost of compression and decompression when reading and writing, but CPU cycles are often cheaper than cloud storage and network bandwidth. Indeed, for archival data, the bulk of the data's cost will lie in its long-term storage rather than any CPU cycles used to compress or decompress it.

In addition to saving space, users would like to keep their data private. There is some risk to privacy when storing data in the cloud and encryption helps mitigate that risk by keeping the data hidden away from prying eyes. As part of our scheme, we present the first provably secure compression algorithms that compress the data while providing strong guarantees of secrecy for the data. The secure compression algorithms—called the *Slow Squeeze* and *Fast Squeeze* ciphers, or simply the

[22] In some rough benchmarks, we used the `speed` command of the OpenSSL 1.0.1i library to to compare the speed of 2048-bit RSA signing and verification to HMAC using MD5 on an Intel Core i5 processor. For RSA, OpenSSL was able to perform 760 signs and 24800 verifications per second (using the public exponent $e = 65537$). For HMAC-MD5, with a 256-byte input—which is the same size input to 2048-bit RSA—OpenSSL was able to perform 1345132 HMAC computations per second.

squeeze ciphers—are based on the LZW compression algorithm [165], a widely used algorithm found in the Unix `compress` utility [66], the GIF image standard [67], and is included in the PDF standard as an optional compression scheme [43]. Fast Squeeze can provide compression ratios and speeds comparable to standard algorithms such as plain LZW, gzip, and bzip2. Alternative paradigms for providing secure compression is an area of future work.

For our storage system, the usage of compression and error correcting codes presents some challenges.[23] If an entire file is compressed and then encoded, subsequent changes to the file may change the compressibility, and hence final size, of the compressed file. This poses challenges for the ECCs previously mentioned since they cannot handle insertions and deletions in the middle of the codeword (though, they can support appending). In this case, the entire (updated) file would need to be re-encoded before being uploaded to the cloud. Another difficulty is that few compression algorithms, and none of the popular ones (including LZW), provide for random access to data in the compressed stream. Rather, one must start at a particular point—the beginning of the stream for LZW, gzip, and squeeze, and at a 400/600/900KB boundary for bzip2—and decompress from there. However, using compression can be made more amenable to reads and writes by simply dividing files into (possibly fixed-sized) blocks and compressing, encrypting, and encoding them separately. This allows for a much finer granularity for read and write operations and make the usage of compression more tenable for warmer (i.e., non-archival) data, but the client must track the mapping between plaintext offsets and compressed ciphertext offsets. However, this is a cost that may be outweighed by the increased efficiency of the scheme for small reads and writes.

1.2.6 Summary of Enhanced Scheme.

The enhanced scheme given in this section has several desirable properties. First, it can tolerate the loss and/or corruption of at most $n - k$ servers, as before. Second, the scheme is publicly verifiable, if we replace the MACs in the Falcon code with a public-key signature scheme (with an increase in storage and computation overhead). Third, the (uncompressed) storage overhead is $(\frac{1}{R} - 1)|F|(1 + s/t)$, where the $(\frac{1}{R} - 1)|F|$ term is from the redundancy added by the ECC (with rate R), and the $(1 + s/t)$ term is the overhead for each symbol from the appended MAC in the Falcon code where each

[23] If we combine compression, even secure compression, with a Falcon code, the compression *must* be applied prior to any encoding. All symbols have a fixed size and variations in their compressed length—e.g., by compressing individual code symbols—can leak information about the contents and enable the targeted erasure attack. See [80] for a thorough discussion of the leakage inherent to schemes combining compression and encryption.

Table 1.2: The efficiency of our enhanced secure storage scheme for various operations. We encode a file F, consisting of $|F|$ bytes, using a Falcon code with k message symbols (each t bytes in size) per stripe. As with a fixed-rate code, we define $R = k/n$. The MAC/signature on each symbol is s bytes in size.

	Computation cost	Communication cost
Bulk put	$O(\frac{1}{R}(\|F\|/t))$	$O(\frac{1}{R}\|F\|(1+s/t))$
Bulk get	$O(\frac{1}{R}(\|F\|/t))$	$O(\frac{1}{R}\|F\|(1+s/t))$
Small read	$O(k)$	$O(k(t+s))$
Small write	$O(n)$	$O(n(t+s))$
Repair	$O(n)$	$O(n(t+s))$

MAC/signature is s bytes and a symbol is t bytes. Finally, the client must store $O(1)$ secret keys for the Falcon code.

We summarize the costs of our storage scheme in Table 1.2 (cf. Table 1.1). Recall that to have the same code rate R as the Reed-Solomon code, we have the Falcon encoder add a $(\frac{1}{R} - 1)$-fraction of parity symbols so that the ratio of input and output lengths is R. We also use the Falcon code in the LT coding step of a Raptor code, giving us a secure linear-time encoding and decoding scheme. We let $n = k/R$ be the number of symbols generated by the Falcon encoder in a stripe. We note that if we use Fast Squeeze and achieve a compression ratio of ρ, this reduces the overhead for reading and writing to a ρ-fraction of the original amount. But this reduction is independent of the ECC used and is equally applicable to the basic scheme, so we omit the savings here.[24]

The efficiencies given in Table 1.2 are not necessarily comparable to those in Table 1.1. In particular, the small read and repair costs are quite different since Falcon codes have a high lower-bound on the value of k. As an example, if each stripe symbol is a single byte, then to read a single byte, the Falcon code must fetch at least 10000 bytes from the server. In the original scheme, if both a block and a hash are 16 bytes, then the Merkle tree for a symbol would need to have at least $(10000 - 16)/16 = 624$ hashes on a single path to fetch the same amount of data. That is, the symbol would need to have at least 2^{624} blocks (or 2^{628} bytes) for 10000 bytes to be transferred to read a single block. However, despite the loss in efficiency for small reads and writes, the savings in the bulk upload and download costs are substantial. The encoding and decoding of the Falcon code require only a *linear* amount of time, instead of quadratic (see Figure 4.5 in Section 4.7 of Chapter 4 for a comparison of the throughput of Reed-Solomon and Falcon codes).

[24]The overhead from using Fast Squeeze is quasi-linear in the input stream length. Chapter 2, Section 2.4 gives a detailed analysis of the efficiency of Fast Squeeze where we show that both compression and decompression take $O(m \log d)$ time to compress m characters, where d is the size of the dictionary.

CHAPTER **Two**

Squeeze Cipher: Secure Compression

2.1 Introduction

Individuals and organizations are producing and retaining data in volumes that are unprecedented. One technique widely used to reduce the footprint of this data is to simply filter it through a compression algorithm before transmission or storage. Indeed, many popular file systems utilize transparent file compression, including Apple's HFS+ [6], Microsoft's NTFS [112], and Oracle's ZFS [19]; additionally, both the standard internet protocol HTTP [41] and Google's own SPDY [53] support compression. Concurrent with the accumulation of mountains of data, data thefts have been increasing in frequency, sophistication, and aggression. Encryption plays a key role in securing data from these attacks both while at rest (e.g., in NTFS, HFS+, and ZFS) and while in transit (e.g., TLS [35] and SPDY), and applying compression before encryption is a natural choice. However, since compression changes the characteristics of the text being encrypted, we must take care to precisely describe and analyze the security provided by the combination of these primitives.

In this work we present a theoretical framework for analyzing and proving the security of combined compression and encryption schemes. We also provide the first provably secure compression algorithms that compress the data while providing strong privacy guarantees for the data. These secure compression algorithms—called the *squeeze* ciphers—are based on the LZW algorithm [165], a widely used compression scheme found in the Unix `compress` utility [66], the GIF image standard [67], and is included in the PDF standard as an optional compression scheme [43]. The first construction, called Slow Squeeze, possesses strong theoretical properties but is impractical. The second construction,

called Fast Squeeze, is both simple and theoretically sound while providing compression ratios and speeds comparable to standard algorithms such as gzip and bzip2.

General Problems Combining Compression and Encryption. Unfortunately, there are some drawbacks to combining compression and encryption together in a naïve fashion. First, the system must perform two passes over the data: compressing then encrypting. Since these operations are often chained together, it would be ideal to achieve a compressed and encrypted output in a single pass over the data. While compression and encryption can be set up to operate in a pipeline (i.e., data is encrypted in a streaming manner as it comes out of the compression routine), there are still two passes over the data occurring.

More troubling, though, is the fact that by combining these primitives together it becomes impossible to achieve semantic security: the standard, minimal notion of security for a cipher. Moreover, it is impossible to achieve semantic security even when encrypting with a cipher secure against adaptive chosen-ciphertext attacks, the gold standard of security for encryption. This failure occurs because in the game-based definition of semantic security an adversary $\mathcal{A}$ chooses two messages, one of which (chosen at random) will be encrypted and its ciphertext returned. $\mathcal{A}$ then tries to determine which message was encrypted. In this setup, $\mathcal{A}$ is permitted to select *any* two messages that will be encrypted. In particular, $\mathcal{A}$ may select one message that is highly compressible and another that is highly *in*compressible, resulting in different length ciphertexts (since the input is compressed then encrypted).

It is also worth noting that the length of a (compressed) ciphertext can leak information to an attacker, e.g., the level of compressibility of the file can gives hints about both the content and structure of the file. See [80] for a full discussion of the information leakage of a general composition of compression and encryption.[1] This leakage is inherent to any scheme that combines compression and encryption, and where the attacker has knowledge of the input and output lengths. We do not address this leakage in this work. See Section 2.8.3 for an overview of past work on length-hiding encryption, which seeks to limit the leakage from compression.

Genesis. This work was born from the observation that compressed data and encrypted data, ostensibly, look very similar. In particular, a good compression algorithm shrinks the input so that the output is as long as the "information content" of the input and, moreover, the output appears

[1] Note that this side-channel is exploited in real systems, e.g., [139].

close to uniformly distributed (ignoring accompanying header information). The output of a good encryption algorithm, in comparison, is pseudorandom: i.e., the output is indistinguishable from a uniformly random string of the same length. Given this apparent similarity, how much privacy is provided by compressing data? Can data compression serve as a kind of "poor man's" encryption? Unfortunately, the answers to these questions are "little to none" and "no."

Data compression provides little real privacy simply because compression algorithms are deterministic and can be inverted by anyone. Even attempts to obfuscate some of the information needed for decompression (e.g., the prefix tree used in Huffman encoding) do not provide security against an adversary that knows some statistics about the input message (e.g., the language used and its associated character frequencies). Thus, the question becomes, can we modify the compression algorithm (in a light-weight way), casting away the determinism, so that the output of the algorithm *itself* provides strong confidentiality for the input message? In this chapter we answer this question in the affirmative.

Contributions. Our contributions to this topic are several. First, in this work we present a new definition of semantic security for combined compression-encryption systems (which was not previously formalized). This definition generalizes the standard notion of semantic security in a natural and intuitive way and, furthermore, also provides corresponding definitions for non-adaptive and adaptive chosen-ciphertext security (CCA-1 and CCA-2, respectively). In addition to establishing a theoretical framework, we provide two constructions that *provably* achieve security in this framework. Moreover, one construction is quite efficient, achieving compression ratios and speeds comparable to standard compression and encryption algorithms. We demonstrate this with a thorough experimental evaluation on multiple computer architectures.

Organization. The remainder of this chapter is organized as follows. In Section 2.2, we provide preliminary definitions for tools that will be used throughout the chapter. Section 2.3 details the LZW compression algorithm, which forms the basis for our secure constructions. Section 2.4 details our constructions for secure compression algorithms called the Fast and Slow Squeeze ciphers. Section 2.5 provides a security analysis of the different schemes and proves that they achieve the claimed security properties. Section 2.6 provides an extensive experimental evaluation of the Fast Squeeze cipher showing its practicality. We detail some applications and variants of the cipher in

Section 2.7. Section 2.8 provides a high-level overview of previous work on secure compression and authenticated encryption. And finally, we discuss future research directions in Section 2.9 and conclude in Section 2.10.

2.2 Preliminaries

In this section we will provide basic definitions of all the primitives we use, as well as several different definitions of security for the various primitives that we employ. First, however, we define the notation that we will use throughout this chapter.

2.2.1 Notation

We denote the security parameter by λ and the empty string by Λ. Let PPT be short for "probabilistic polynomial-time." If Alg is a PPT algorithm, let $[\mathsf{Alg}(\pi)]$ denote the set of all possible outputs of Alg when run on parameters π. Let $x \xleftarrow{R} S$ denote sampling x from the set S uniformly at random. Let $x \leftarrow \mathcal{D}$ denote sampling x according to distribution $\mathcal{D}$. There is some notational overloading as we use $\leftarrow$ for assignment in algorithm listings, but its meaning will clear from the context or clearly marked in a comment. Let $\circ$ denote concatenation and $l(s)$ and $|s|$ the length of a string s. Let $\mathcal{C}$ be the space of ciphertexts and $\mathcal{M}$ the space of messages. Let $\mathcal{P}(n)$ be the set of all permutations of binary strings of length n. We use $\mathbb{N}$ to denote the set of positive integers. For a keyed function $f(k, \cdot)$, we will often denote this as $f_k(\cdot)$.

2.2.2 Basic Definitions

First, we provide a definition of data compression. At a high-level, data compression seeks to conserve storage capacity and network bandwidth by transforming data into a more compact form. More concretely, data compression is an encoding of a sample from a data source such that, in expectation, an encoded message is no longer than the original data. Since it is not possible to achieve compression in all cases (i.e., for all binary strings—for example, there are 2^k k-bit binary strings and only $2^k - 1$ strings of length less than k), we concern ourselves with compressing specific subsets of messages and seek to achieve compression in expectation.

Definition 2.1. Let $\mathcal{M} \subseteq A^*$ be a space of messages over alphabet A with associated probability distribution $\mathcal{D}$ over $\mathcal{M}$. An *encoding* for $\mathcal{M}$ is a map $C : \mathcal{M} \to \mathcal{S}$, for some set of binary strings $\mathcal{S}$.

If $\mathcal{S} = \{0,1\}^*$, then C is called a *variable-to-variable* code. If $\mathcal{S} = \{0,1\}^n$, for some $n \in \mathbb{N}$, then C is called a *variable-to-fixed* code.[2] An element in the image of $\mathcal{M}$ is a *codeword*. If any sequence of codewords can be parsed unambiguously, then C is called *uniquely decodable*. □

We only consider uniquely decodable codes. Given a probability distribution D over a message space $\mathcal{M}$, let $p(m)$ be the probability of sampling message m from $\mathcal{M}$ according to $\mathcal{D}$ and let $l(m)$ be the length of m. The *average length* of a code is defined as,

$$L(C) = \sum_{m \in \mathcal{M}} p(m) l(C(m)).$$

Ideally, we want to design the encoding C to obtain,

$$L(C) \leq \sum_{m \in \mathcal{M}} p(m) l(m).$$

If so, then we will refer to C as a *compression function*. We call the associated inverse map from the set of codewords to $\mathcal{M}$ the *decoding function* and denote it with D. If C is a compression function, then we call D the *decompression function*.

We define a *keyed compression function*, also referred to as a *compressing cipher*, to be a compression function that takes an additional parameter k, such that k is required to correctly decompress the input.

Definition 2.2. A *keyed compression function* over a message space $\mathcal{M} \subseteq A^*$ and ciphertext space $\mathcal{C}$, is a triple of (possibly probabilistic) polynomial-time algorithms $(\mathsf{Gen}, \mathsf{Encode}, \mathsf{Decode})$ where,

- Gen takes as input 1^λ and outputs a key k.
- Encode is a compression function that takes as input a key k and a plaintext $m \in \mathcal{M}$ and outputs a (compressed) ciphertext $c \in \mathcal{C}$.
- Decode takes as input a key k and a ciphertext $c \in \mathcal{C}$ and outputs the plaintext $m \in \mathcal{M} \cup \{\perp\}$.

We require that for all $k \in [\mathsf{Gen}(1^\lambda)]$, for all $m \in \mathcal{M}$, $\mathsf{Decode}(k, \mathsf{Encode}(k, m)) = m$. □

Any plain compression scheme trivially satisfies this definition by having the key-generation algorithm Gen always output $\perp$ and by letting Encode and Decode be the usual compression and decompression functions. Of course, such a set up gives little, if any, privacy. Note, also, that any length-preserving cipher applied to a prefix-free set of messages also satisfies this definition since the

[2] If $\mathcal{M} \subseteq A^k$, for some $k \in \mathbb{N}$, then $C : \mathcal{M} \to \mathcal{S}$ is a *fixed-to-variable* code if $S = \{0,1\}^*$. If $S = \{0,1\}^n$, then C is a *fixed-to-fixed* code.

identity function is a valid compression function for that set.[3] This definition of a keyed compression function is a particular instance of a *symmetric encryption scheme* (or *private-key encryption scheme*). Such a scheme is a triple of algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, where Gen is the key generation algorithm (where the key is kept secret), and Enc and Dec are encryption and decryption, respectively, and operate in the expected way—i.e., by taking the secret key and a message/ciphertext as input and encryption/decrypting it.

As part of the construction of our compressing cipher, we will use a *pseudorandom permutation* (PRP) in a key-derivation process that incorporates a nonce (or initialization vector) into each encryption. Let $\mathcal{O}_f$ be an oracle for the function $f_k(\cdot)$, where $k \leftarrow \{0,1\}^\lambda$ and f is a permutation of $\{0,1\}^n$. We do not give the adversary $\mathcal{A}$ access to an oracle for f^{-1} and hence only require a "weak" PRP. The cryptographic definitions used and the security analysis performed in this work will be concrete, but for simplicity we will often omit the exact security of a given primitive.

Definition 2.3 (Pseudorandom Permutation)**.** A keyed permutation $f : \{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}^n$ is (t, q, ε)-*indistinguishable*, if for all PPT distinguishers $\mathcal{D}$ running in time t and making q queries to $\mathcal{O}_f$, distinguishes f from a permutation $\pi \xleftarrow{R} \mathcal{P}(n)$ with probability at most $\frac{1}{2} + \varepsilon$. □

If we relax the restrictions on $\mathcal{D}$ and only require that t and q be polynomially-bounded, then if ε is negligible, we say that f is *computationally indistinguishable* from a random permutation, or just *computationally secure*. The typical example of a pseudorandom permutation is a block cipher, such as AES.

Another tool used in our constructions is a *pseudorandom generator* (PRG). A PRG is a deterministic algorithm that given a short, random seed will produce a long stream of pseudorandom bits. A PRG is *secure* or *cryptographically strong* if, given a random seed, its output is indistinguishable (in polynomial time) from truly random bits.

Definition 2.4 (Pseudorandom Bit Generator)**.** A bit generator G is a pair of algorithms $(\mathsf{Gen}, \mathsf{F})$, where Gen is a PPT algorithm that on input 1^λ outputs a seed $s \in \{0,1\}^\lambda$, and $\mathsf{F} : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^*$. Note that the second parameter of $\mathcal{F}$ specifies the number of bits to output, i.e., $|\mathsf{F}(s,i)| = i$. Moreover, $\mathsf{F}(s,i)$ is a prefix of $\mathsf{F}(s,i+1)$. G is (t, b, ε)-*indistinguishable* if for all polynomial-time distinguishers $\mathcal{D}$, running in time t and given b bits of input, distinguishes $\mathsf{F}(s,b)$ from a random string $s \in \{0,1\}^b$ with advantage at most ε over $\frac{1}{2}$. □

[3] If the set is not prefix-free, then it is not possible to unambiguously parse a sequence of codewords.

We say that a pseudorandom generator is *secure* if both t and b are polynomially bounded, and ε is negligible function of the security parameter λ. We will typically refer to G as *pseudorandom* or a *pseudorandom generator* rather than specify its exact, concrete indistinguishability. For notational simplicity, we will denote $\mathsf{F}(s, b)$ by $G(s)$ and let b be implicit from the context. Note that one could define an adaptive variant of G where F is a stateful algorithm and outputs its bits one-by-one (or k at a time). The distinguisher would then take the bits from F dynamically; though, the total number of bits would still be bounded by b.

Since we would like our scheme to be able to encrypt multiple messages, we require the PRG to be given a fresh seed for each encryption. Normally, a distinguisher $\mathcal{D}$ for G is given a string s of length at most b from an oracle $\mathcal{O}$ and must guess whether s is random or pseudorandom. But, since we allow our adversary to make multiple encryption queries we must have a PRG that is secure even if it is reseeded multiple times. Therefore, we allow $\mathcal{D}$ to issue "reseed" requests to $\mathcal{O}$. If $\mathcal{O}$ is simply G, then it is initialized with a new random seed and a new sample is given to $\mathcal{D}$. If $\mathcal{O}$ produces random bits, then $\mathcal{D}$ is just given a new random string.[4] A PRG that is secure in this setting we call *reseedably-indistinguishable*.

Definition 2.5. A PRG G is (t, r, b, ε)-*reseedably-indistinguishable* if for all PPT distinguishers $\mathcal{D}$, running in time t, making at most r reseed requests, and receiving at most b bits as input per seeding, succeeds in distinguishing the output of G from random with probability at most $\frac{1}{2} + \varepsilon$. □

We will assume that G can produce a super-polynomial number of bits and that, in addition to the sample s, $\mathcal{D}$ may request additional, subsequent bits from the output of G. For this reason, in the rest of this work we will omit the b parameter. As before, relaxing the restrictions on $\mathcal{D}$ and letting t and r be just polynomially bounded, if ε is negligible then we say that such a PRG is *computationally secure*. Note that requiring that a PRG to be reseedably-indistinguishable does not significantly change its strength. In Appendix A we formally prove the polynomial-equivalence of reseedable-indistinguishability and standard indistinguishability through a straightforward hybrid argument. We summarize this result with the following lemma.

Lemma (Indistinguishability Equivalence)**.** *Let G be a (t, b, ε)-indistinguishable PRG, then G is $(t - r, r + 1, b, \varepsilon r^2)$-reseedably-indistinguishable.*

[4] This models typical reseeding of a PRG in practice, i.e., generate a new random seed and forget the old one.

Finally, another basic tool that we use is a message authentication code (MAC). Briefly, a message authentication code is a triple of algorithms $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Verify})$ that, given a key k and a message m, produce a tag t for the message.

Definition 2.6 (Message Authentication Code)**.** A *message authentication code* (MAC) for a message space $\mathcal{M}$ and tag space $\mathcal{T}$, is the triple of algorithms, $M = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Verify})$ where,

- Gen is a PPT algorithm that on input 1^λ outputs a key $k \in \{0,1\}^\lambda$.
- Mac is a PPT algorithm that on input $k \in \{0,1\}^\lambda$ and $m \in \mathcal{M}$ outputs a *tag* $t \in \mathcal{T}$.
- Verify is a deterministic algorithm that on input $k \in \{0,1\}^\lambda$, $m \in \mathcal{M}$, and $t \in \mathcal{T}$ outputs a bit $b \in \{0,1\}$.

We require that for all $k \leftarrow \mathsf{Gen}(1^\lambda)$ and all $m \in \mathcal{M}$, $\mathsf{Verify}(k, m, \mathsf{Mac}(k, m)) = 1$. □

Intuitively, we would like a MAC scheme to be "unforgeable" so as to provide strong assurances of message authenticity and integrity. The strongest notion of unforgeability is the following: no PPT adversary can forge a signature for *any* message, even one of its own choice (this is called *existential unforgeability*). In the below definition we give the adversary $\mathcal{A}$ access to a Mac oracle $\mathcal{O}_k$ that on input m produces $t = \mathsf{Mac}(k, m)$. After a number of queries, bounded by some q, $\mathcal{A}$ outputs a message and tag pair (m, t) and "wins" if the MAC verifies for m (i.e., $\mathsf{Verify}(k, m, t) = 1$). We also require that m was not previously queried to the oracle.

Definition 2.7 (Existentially Unforgeable MAC)**.** A MAC scheme M is (t, q, ε)-*unforgeable* if for all PPT adversaries $\mathcal{A}$ running in time t, given access to an oracle $\mathcal{O}_k$ (where $\mathcal{O}_k(m) = \mathsf{Mac}(k, m)$ for some $k \leftarrow \mathsf{Gen}(1^\lambda)$), we have that,

$$P[k \leftarrow \mathsf{Gen}(1^\lambda); \mathcal{A}^{\mathcal{O}_k(\cdot)}(1^\lambda) \rightarrow (Q, m, t) : m \notin Q \wedge \mathsf{Verify}(k, m, t) = 1] \leq \varepsilon$$

where Q is the list of oracle queries made by $\mathcal{A}$ and $|Q| \leq q$. The probability is taken over the random coins of A, Gen, and the oracle $\mathcal{O}_k$. □

If t and q are polynomially-bounded, and ε is a negligible function of λ, then we say M is *existentially-unforgeable* or just *computationally-secure*.

2.2.3 Entropy-restricted Semantic Security

When constructing a cipher, we are concerned with "hiding" as much information as possible in the input message m. That is, intuitively, we want that the ciphertext leaks no significant information

about the plaintext. Put another way, we want that given any pair of messages, their respective ciphertexts are indistinguishable.[5] To accomplish this, we give the adversary $\mathcal{A}$ access to an encryption oracle that is instantiated with a random key. After some number of queries, $\mathcal{A}$ outputs a pair of message m_0 and m_1 (the "challenge messages"). One of the messages is chosen at random and encrypted, and the resulting ciphertext is given to $\mathcal{A}$. The task of $\mathcal{A}$ is to guess which challenge message was encrypted, and we say that the scheme is secure if $\mathcal{A}$ cannot do this with non-negligible advantage over 1/2. This notion is formalized in the following game.[6]

1. A key k is generated by $\mathsf{Gen}(1^\lambda)$.
2. The adversary $\mathcal{A}$ is given 1^λ as input, has oracle access to $\mathsf{Enc}_k(\cdot)$, and outputs a pair of message m_0, m_1 of the same length.
3. A random bit $b \leftarrow \{0,1\}$ is chosen, and $c \leftarrow \mathsf{Enc}_k(m_b)$ is given to $\mathcal{A}$.
4. $\mathcal{A}$ continues to have oracle access to $\mathsf{Enc}(k,\cdot)$ and outputs a bit b'.
5. $\mathcal{A}$ wins the game if $b' = b$.

Security against a PPT adversary in this game is called *chosen-plaintext attack indistinguishability*, denoted IND-CPA. Denote the oracle for $\mathsf{Enc}(k,\cdot)$ by $\mathcal{O}(\cdot)$. We define the *success probability* of $\mathcal{A}$ in the game to be,

$$\mathsf{Succ}^{\text{IND-CPA}}_{\Pi,\mathcal{A}}(1^\lambda) = P[k \leftarrow \mathsf{Gen}(1^\lambda); \mathcal{A}^{\mathcal{O}(\cdot)}(1^\lambda) \rightarrow m_0, m_1; b \leftarrow \{0,1\}; \mathcal{A}^{\mathcal{O}(\cdot)}(\mathsf{Enc}(k, m_b)) \rightarrow b'; b = b'],$$

where the probability is over the random coins of Gen, $\mathcal{A}$, and the oracle $\mathcal{O}$. Also, we define $\mathsf{Adv}^{\text{IND-CPA}}_{\Pi,\mathcal{A}}(1^\lambda) = |\mathsf{Succ}^{\text{IND-CPA}}_{\Pi,\mathcal{A}}(1^\lambda) - \frac{1}{2}|$ to be the *advantage* of $\mathcal{A}$ in the game.

Definition 2.8. A symmetric encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is (t, q, ε)-*IND-CPA secure* if for all PPT adversaries $\mathcal{A}$ running in time t and making at most q oracle queries to $\mathcal{O}(\cdot)$, $\mathsf{Adv}_{\Pi,\mathcal{A}}(1^\lambda)$ is at most ε. □

In considering the security of combining compression and encryption together, as stated before, it is *not* possible for a such a system to be IND-CPA secure. In particular, since the adversary $\mathcal{A}$ may choose arbitrary challenge messages, it can select a high-entropy m_0 and a low-entropy m_1.[7] With these, a keyed compression function will likely produce a much shorter output for m_1 than for m_0, allowing $\mathcal{A}$ to easily distinguish them.

[5] These two formulations were proven equivalent in [48].

[6] The game-based definition is lifted almost verbatim from [77].

[7] We use the term "entropy" *not* in its mathematical sense; rather, we use it as an informal synonym for "information content" and "length when compressed."

Instead of IND-CPA security, we aim for an *entropy restricted* variant. Specifically, we restrict $\mathcal{A}$ so that m_0 and m_1 must both come from the same class of messages $\mathcal{C} \subseteq \mathcal{M}$. We call such as class a *compression class*. A scheme is secure relative to a class $\mathcal{C}$ if the encryptions of any $m_0, m_1 \in \mathcal{C}$ are indistinguishable. For example, if $\mathcal{C} = \mathcal{M}$, then we have normal semantic security. Letting C be a keyed compression function, we can define $\mathcal{C}_{l_1,l_2} = \{m \in \mathcal{M} : l_1 \leq |C(m)| \leq l_2\}$, i.e., m_0 and m_1 compress to lengths within a fixed range. This can be extended to all of $\mathcal{M}$, partitioning the message space into equivalence classes. Note that, given a scheme that is secure when $l_1 = l_2$, we can extend it to the case where $l_1 < l_2$ by simply appending an end-of-file marker to the plaintext and then appending random padding to the ciphertext until all ciphertexts have length l_2.[8] The oracle can ensure that two messages come from the same class by encrypting both and examining the lengths.

This approach is in contrast to the work of [162] where Tezcan and Vaudenay examine the security provided by adding padding to the end of a plaintext message. They consider an adversary that seeks to distinguish the combined padding and encryption of two arbitrary messages that are allowed to be of different lengths (with a bounded difference). That is, implicitly they consider the case where the challenge messages may come from *different* classes (as defined in this work). They prove that *in*security decreases *linearly* with padding length and that selecting the padding length uniformly at random is nearly optimal. As stated above, we only consider indistinguishability of messages *within a single class*.

Note that it can be difficult to ascertain whether or not two messages m_0 and m_1 belong to the same class without compressing the messages themselves (see [61]). With our particular constructions, there exists a public, deterministic algorithm that can compress messages to the exact same length as our ciphers. But, in general, this may not be possible with other schemes. This means that an adversary $\mathcal{A}$ may not know a priori whether or not two messages compress to the same length. In such a situation, the oracle enforces that m_0 and m_1 come from the same class, and if they do not then $\mathcal{A}$ is not penalized (i.e., the query is not "counted").

We define *entropy-restricted IND-CPA security* via a game. The game is nearly identical to the game for IND-CPA security, but we change the requirement on the challenges to be $m_0, m_1 \in C$, for some class $\mathcal{C} \subseteq \mathcal{M}$. Define $\mathsf{Succ}^{\text{ER-CPA}}_{\Pi,\mathcal{A},\mathcal{C}}(1^\lambda)$ to be the probability that an adversary $\mathcal{A}$ succeeds in the ER-CPA game for class $\mathcal{C}$. Similarly, define $\mathsf{Adv}^{\text{ER-CPA}}_{\Pi,\mathcal{A},\mathcal{C}}(1^\lambda) = |\mathsf{Succ}^{\text{ER-CPA}}_{\Pi,\mathcal{A},\mathcal{C}}(1^\lambda) - \frac{1}{2}|$ to be the advantage of $\mathcal{A}$ in the ER-CPA game.

[8] That is, we make the ciphertext self-delimiting from the padding.

Definition 2.9. We say that a symmetric encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is (t, q, ε)-*entropy-restricted CPA secure* for a class of messages $\mathcal{C} \subseteq \mathcal{M}$, if for all PPT $\mathcal{A}$ running in time t and making at most q queries to the encryption oracle $\mathcal{O}$, $\mathsf{Adv}^{\text{ER-CPA}}_{\Pi,\mathcal{A},\mathcal{C}}(1^\lambda)$ is at most ε. □

Often, we will just term this ER-CPA security and omit the specification of (t, q, ε), except when working in a formal context. For simplicity we will only consider the classes of messages $\mathcal{C}_{l_1,l_2}$ where $l_1 = l_2$, since a scheme secure for those classes can be extended to a scheme that is secure for $l_1 < l_2$.

We note that there are two strictly stronger notions of security for a cipher, specifically IND-CCA-1 and IND-CCA-2 (for "chosen-ciphertext attack indistinguishability"). Roughly, the games for IND-CCA-1 and IND-CCA-2 security proceed the same as for IND-CPA security, except the adversary $\mathcal{A}$ is given access to a *decryption* oracle in addition to the encryption oracle. For IND-CCA-1, $\mathcal{A}$ can only query the decryption oracle *before* outputting the challenges m_0 and m_1. After receiving the challenge ciphertext c, $\mathcal{A}$ may only query the encryption oracle. For IND-CCA-2, $\mathcal{A}$ is allowed to query the decryption oracle after receiving c as well, but $\mathcal{A}$ may not query c to the oracle (otherwise IND-CCA-2 security would be impossible to achieve). Here we also define analogous entropy-restricted CCA-1 and CCA-2 notions of security for a class $\mathcal{C} \subseteq \mathcal{M}$, which we will term ER-CCA-1 and ER-CCA-2. The definitions of $\mathsf{Succ}^{\text{ER-CCA-1}}_{\Pi,\mathcal{A},\mathcal{C}}(1^\lambda)$, $\mathsf{Succ}^{\text{ER-CCA-2}}_{\Pi,\mathcal{A},\mathcal{C}}(1^\lambda)$, $\mathsf{Adv}^{\text{ER-CCA-1}}_{\Pi,\mathcal{A},\mathcal{C}}(1^\lambda)$, and $\mathsf{Adv}^{\text{ER-CCA-2}}_{\Pi,\mathcal{A},\mathcal{C}}(1^\lambda)$ are defined analogously.

Definition 2.10. We say that a symmetric encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is $(t, q_1, q_2, \varepsilon)$-*entropy-restricted IND-CCA-X secure* (or $(t, q_1, q_2, \varepsilon)$-*ER-CCA-X secure*) with $X \in \{1, 2\}$, for a class of message $\mathcal{C} \subseteq \mathcal{M}$, if for all PPT adversaries $\mathcal{A}$ running in time t, making at most q_1 encryption queries and making at most q_2 decryption queries, $\mathsf{Adv}^{\text{ER-CCA-X}}_{\Pi,\mathcal{A},\mathcal{C}}(1^\lambda)$ is at most ε. □

We note that any IND-CPA secure cryptosystem combined with a compression pre-processing step is automatically ER-CPA secure, else we could easily create an algorithm to break the IND-CPA security of the scheme. Specifically, if $\mathcal{A}$ breaks the ER-CPA security of the scheme, then we can construct $\mathcal{B}$ which takes each query q made by $\mathcal{A}$, compresses it, and then queries its own oracle. When $\mathcal{A}$ outputs its challenges m_0 and m_1, we know that the compressed lengths are equal and so, once compressed, they are valid challenges for $\mathcal{B}$ to use. After receiving back the encryption of m'_b for some random b, the ciphertext is passed back to $\mathcal{A}$ and $\mathcal{B}$ outputs whatever $\mathcal{A}$ does. Analogous reductions show that an IND-CCA-1 and IND-CCA-2 secure ciphers are ER-CCA-1 and ER-CCA-2 secure (respectively).

Even though ER-CPA security and its variants are new definitions of security, they relate in a simple way to the standard definitions. Firstly, as shown above, being secure in the usual sense implies security in our entropy-restricted context. Secondly, there are many results constructed from IND-CPA/CCA-1/CCA-2 primitives, but, crucially, few of these results depend on the *length* of the input message. Thus, many of these results also hold when considered in the context of ER-CPA/CCA-1/CCA-2 security. This is intuitively true since the restriction that challenge messages have the same length is to ensure that the *ciphertexts* have the same length, i.e., the length of the object given to $\mathcal{A}$ does not divulge information about the input message. As an example, the results on ciphertext unforgeability in [78], which we discuss next, also hold in our entropy-restricted context.

2.2.4 Ciphertext Unforgeability

Katz and Yung in [78] define the notion of "ciphertext unforgeability" where it is infeasible for any PPT algorithm to generate a valid ciphertext without access to the encryption key. Intuitively, if a cipher possesses this property, then a decryption oracle becomes "useless" to the adversary. Thus, coupling unforgeability with a notion of privacy (e.g., semantic security) one can gain much stronger security, such as resistance to chosen-ciphertext attacks. Katz and Yung define several distinct notions of unforgeability, but we will concern ourselves with the strongest definition: *existential unforgeability*. We reproduce their definition here but with a slight change in terminology. What they define as $(t, q, b; \epsilon)$-*secure* we call (t, q, ε)–*unforgeable*, since we will be working with multiple notions of security in this chapter and we do not quantify the number of bits b received as input to the adversary (the reasoning for this is explained below).

Definition 2.11. Let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme, and let $\mathcal{A}$ be a PPT adversary. Define:

$$\mathsf{Adv}^{\text{exist}}_{\mathcal{A},\Pi} = P[k \leftarrow \mathsf{Gen}(1^\lambda); y \leftarrow \mathcal{A}^{\mathsf{Enc}_k(\cdot)} : \mathsf{Dec}_k(y) \neq \perp].$$

We require that $\mathcal{A}$ has never received y from its encryption oracle. We say that Π is (t, q, ε)-*unforgeable*, if for any PPT adversary $\mathcal{A}$ which runs in time at most t and asks at most q queries, we have $\mathsf{Adv}^{\text{exist}}_{\mathcal{A},\Pi} \leq \varepsilon$. □

They also prove that this existential unforgeability coupled with IND-CPA security implies IND-CCA-2 security (see Theorem 1 in [78]). Observing that their proof is independent of the message length, we state, with just a high-level proof, an analogous entropy-restricted version of their result.

Lemma 2.1. *Let* Π *be an encryption scheme that is* $(t_1, q_1, \varepsilon_1)$*-unforgeable and* $(t_2, q_2, \varepsilon_2)$*-ER-CPA secure. Then* Π *is* $(t', q'_e, q'_d, \varepsilon')$*-ER-CCA-2 secure, where* $t' = \min\{t_1, t_2\}$*,* $q'_e = \min\{q_1 - 1, q_2\}$*, and* $\varepsilon' = q_d\varepsilon_1 + \varepsilon_2$*.*

The proof of this proceeds identically to that of the original paper, with the exception that challenge messages must belong to the same compression class. We note two differences between the statement of this lemma and Theorem 1 in [78]. First, they specify that plaintexts have length l, while we allow plaintexts to be of arbitrary length (though the encryptions of two challenges messages must be the same length). Secondly, in addition to running time and the number of oracle queries, they quantify an upper-bound on the number of bits given as input to $\mathcal{A}$ (from the encryption queries). In their scheme, if Π is existentially unforgeable with at most b bits given to $\mathcal{A}$ and Π is IND-CPA secure with at most b_e bits given to $\mathcal{A}$, then it is IND-CCA-2 secure receiving at most $\min\{b - l, b_e\}$. Since we allow plaintexts to have arbitrary (but polynomially-bounded) length, we cannot give an upper bound on the number of bits received by $\mathcal{A}$, and, hence, omit any such bound.

2.3 LZW Compression

In [165], Welch first presented his modification of the older LZ78 compression algorithm [174]. His algorithm, known as LZW, is a dictionary-based compression algorithm where the dictionary is dynamically constructed as the input is processed, as in LZ78. The dictionary D in LZW is initialized with all single-character strings in the input alphabet while LZ78 starts with an empty dictionary. In each iteration of LZW, the next characters are scanned and the algorithm matches the longest prefix p of the input that is in D—i.e., $p \in D$ but $p \circ c \notin D$, where c is the next character. The index of p in D is output and the string $p \circ c$ is added to D. The algorithm then repeats this process on the remaining input starting with the next character c until all input is consumed. This is shown in Figure 2.1 and detailed in Algorithm 2.1. Contrast this with LZ78 where at each iteration, the compression algorithm outputs the index of p as well as the non-matched character c. This allows LZW to achieve better compression than LZ78 since LZW removes these "uncompressed" single characters from the output stream and folds them into the matched strings.[9]

Decompression works in much the same way but in reverse. Namely, the algorithm reads in the next dictionary index, looks up the corresponding entry, and outputs the associated string. Decompression

[9] This has the added benefit of also simplifying the parsing of the compressed data stream.

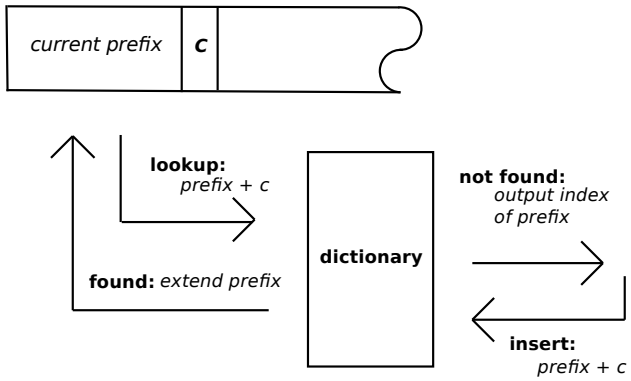


Figure 2.1: The LZW compression algorithm.

is detailed in Algorithm 2.2. Note that there is a special case that must be handled. If the input has a string $c \circ w \circ c \circ w \circ c$, where $c \circ w$ is in the dictionary, then the compression algorithm will match $c \circ w$ and insert $c \circ w \circ c$ into the dictionary. The next string matched is then $c \circ w \circ c$ and its index is output. But, at the receiver, the second index received refers to an *empty* entry in the dictionary. We know, however, that if the empty index points to the next available cell (where the new entry will be placed), then it must be that we have hit this special case. However, if the index does not point to the next available cell then the data has been corrupted and we have a decoding error.

The dictionary used can be of fixed size or it can grow dynamically. The former requires either some sort of deterministic eviction policy or freezing the dictionary when it becomes full. Growing the dictionary dynamically requires that the encoder and decoder grow the dictionary at the same time. Typically, the dictionary is doubled in size (and outputs become one bit longer) when the dictionary is filled. This approach is most common as it gives better compression, using fewer bits early on in the encoding process. We note that our cipher is compatible with *any* dictionary management scheme that is not dependent on the order of the entries in the dictionary (e.g., least-recently-used).

The original LZW paper [165] also describes a space optimization to decrease memory usage of the dictionary.[10] Specifically, each entry consists of a prefix that is found elsewhere in the dictionary with a single character appended (with the exception of the entries consisting of single letters). The optimization is to simply store the index of the prefix and the single character extending the prefix. This greatly reduces the space consumed by the dictionary, but requires a few changes to the decompression algorithm. We refer the reader to the original paper [165] for details.

[10]This space optimization is also possible in LZ78.

Algorithm 2.1 The LZW compression algorithm.

Input: Character stream I
Output: Stream of dictionary indices

Initialize dictionary D to contain all single-character strings
prefix $\leftarrow \Lambda$ ▷ The current prefix of the input
free $\leftarrow$ number of single-character strings ▷ The index of the first free entry in D
while there is more input in I **do**
    Read next character c
    if *prefix* $\circ\, c$ in D **then**
        prefix $\leftarrow$ *prefix* $\circ\, c$ ▷ Extend *prefix* with c and continue processing input
    else
        Output index of *prefix* in D ▷ *prefix* $\circ\, c$ not in D: add it to D
        D[*free*] $\leftarrow$ *prefix* $\circ\, c$
        free $\leftarrow$ *free* $+ 1$
        prefix $\leftarrow c$ ▷ Continuing processing the input starting at c
    end if
end while
if *prefix* $\neq \Lambda$ **then** ▷ Output index of the remaining input
    Output index of *prefix* in D
end if

2.4 Squeeze Ciphers

To turn the LZW algorithm into a cipher, we change how the dictionary D is managed; in particular, we and randomize the order of the entries. We employ two different randomization schemes, each utilizing a PRG as the source of randomness. In the first scheme, after outputting the index of the matched string, we randomly permute all of D. This ensures that at each step the index of the next matched string will be uniformly distributed among all possible values (and hence, achieves perfect secrecy[11] when using truly random bits). For the second scheme, D is randomly permuted after initializing it with all single-character strings and then is incrementally re-randomized by *partially permuting* the dictionary D after making a match. Intuitively, much of the randomness in the permutation of D is left *unused* and can be *reused* on the next iterations, provided that some additional randomness is injected to restore the random permutation. This is detailed below.

In both schemes, a nonce ℓ is given as an input parameter and is then combined with a master secret key to generate a session key for the pseudorandom generator. Since the PRG is used to determine the ordering of entries in the dictionary, we do not want the encoding of one message to leak information about any other message. To derive the ephemeral key, we apply a pseudorandom permutation $f : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$—where λ is the security parameter—to the nonce ℓ to

[11] *Perfect secrecy* is also known as *information theoretic security*. Any scheme with perfect secrecy is secure even from an adversary with *unbounded* computational resources.

Algorithm 2.2 The LZW decompression algorithm.

Input: Sequence of indices I
Output: Stream of characters or the error symbol $\perp$

Initialize dictionary D to contain all single-character strings
$c \leftarrow \Lambda$ ▷ First character of next output string
$prev \leftarrow \Lambda$ ▷ The previously output string
$free \leftarrow$ number of single-character strings
while there is more input in I **do**
    Read next index i
    if $D[i]$ is defined **then** ▷ i is a valid index in D
        $c \leftarrow \text{HEAD}(D[i])$ ▷ Take the first character of $D[i]$
        $D[free] \leftarrow prev \circ c$ ▷ $prev \circ c$ was the value inserted after encoding $prev$
        $free \leftarrow free + 1$
    else if $D[i]$ is not defined and $i = free$ **then** ▷ Special case: input was $c \circ w \circ c \circ w \circ c$
        $c \leftarrow \text{HEAD}(prev)$ ▷ The first character of $prev$ is c
        $D[i] \leftarrow prev \circ c$ ▷ We're decoding $prev \circ c$
        $free \leftarrow free + 1$
    else
        Output $\perp$ and exit ▷ Decoding failed: invalid index
    end if
    Output $D[i]$ ▷ Finally: output $D[i]$ and update $prev$
    $prev \leftarrow D[i]$
end while

compute the seed $f_k(\ell) = s$, where $k \in \{0,1\}^\lambda$ is the master secret key. The nonce is prepended to the output so that receiver can properly initialize G for decryption. We use a PRP for key derivation to simplify the security analysis: using it ensures that each session key is both uniformly distributed and unique. In practice, any secure key-derivation function (such as PBKDF2 [74]) may be used.

2.4.1 Slow Squeeze: Simple, Secure LZW Compression

As stated previously, to turn LZW into a cipher, the simplest solution is to randomly permute the entire dictionary after each iteration. This ensures that whatever the next substitution may be, the output value is uniformly distributed among all possible values.[12] This small modification to LZW, however, is not enough to prevent all attacks (e.g., manipulation of the ciphertext). In particular, as described, all *prefixes* of a ciphertext are themselves valid ciphertexts, allowing for trivial truncation attacks. Such attacks can be thwarted by appending a special "end-of-file" (EOF) symbol to the plaintext message before encrypting. If, when decrypting, the input ends before the EOF has appeared in the decrypted stream (or if the EOF was encountered before the end of the input) then the decryption algorithm fails. This secure LZW algorithm, which we call the Slow

[12] We note that this scheme is entirely compatible with the dynamically-growing dictionary described earlier.

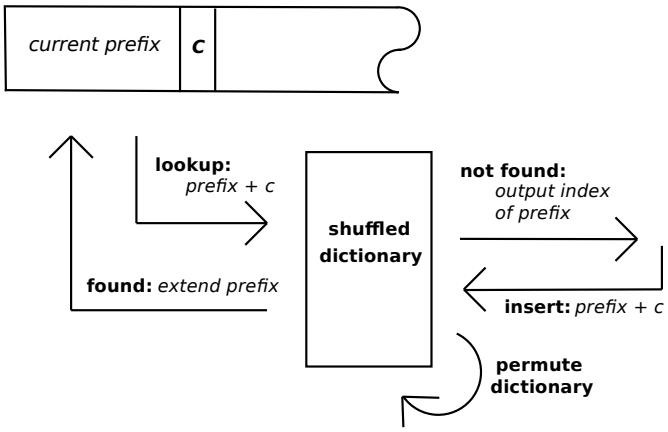


Figure 2.2: The Slow Squeeze secure compression algorithm.

Squeeze cipher—Fast Squeeze is detailed later—is shown pictorially in Figure 2.2. The encryption algorithm is detailed in Algorithm 2.3 and the decryption algorithm is detailed in Algorithm 2.4. As stated previously, Slow Squeeze utilizes a PRP f for session key derivation and a PRG G for its source of randomness. Both of these functions are public parameters to the scheme.

Note that if we limit the load of the dictionary to α, then we do not need to permute the entire dictionary (i.e., we do not need to "move" the empty cells). We can store the dictionary implicitly and map strings to their indices (e.g., via a hash table) and ensure that there are no "collisions" of numbers. This also provides an immense efficiency gain when α is very small (say, $\alpha \leq 2^{-\lambda}$) since we can use far fewer pseudorandom bits (on the order of the number of *used* entries in the dictionary). We will show that this scheme achieves ER-CCA-2 security in Section 2.5.

Efficiency. LZW compression is efficient, taking $O(n)$ time (where n is the input size) since, with a dictionary that performs constant-time look-ups, it performs just a constant amount of work at each step. (For example, a prefix tree or a hash table would allow for constant-time look-ups.)For Slow Squeeze, the compression and decompression functions can also use these efficient data structures for look-ups and processing the input/output characters. However, at each step, Slow Squeeze must permute the entire dictionary. If the dictionary has total size d and indices are of length l, then generating the permutation takes $O(dl)$ time using the Fisher-Yates algorithm[13] (see [42] and [82]) and applying the permutation also takes $O(d)$ time. This gives a total compute time of $O(ndl)$.

[13]Typically, descriptions of the Fisher-Yates algorithm implicitly assume that the PRG can output a random integer in $\mathbb{Z}_d$ in constant time.

Algorithm 2.3 The Slow Squeeze Cipher: a simple, secure LZW compression algorithm.

Input: Key k, nonce ℓ, and input stream I
Output: Encrypted, compressed output stream O

Initialize dictionary D with all single-character strings and EOF symbol
Set $s \leftarrow f(k, \ell)$ and seed PRG G with s ▷ f is a PRP
Output ℓ ▷ G is a PRG
Generate a permutation π using G and apply it to D
Set *prefix* $\leftarrow \Lambda$ and $c \leftarrow \Lambda$
while there is more input in I **do** ▷ Find longest prefix of I that is in D
    Read next character c
    if *prefix* $\circ\, c$ in D **then**
        prefix $\leftarrow$ *prefix* $\circ\, c$
    else ▷ *prefix* is the longest prefix of the input that is in D
        Output index of *prefix* in D
        Insert *prefix* $\circ\, c$ into D in the first free position
        Generate a new permutation π with G and apply it to D
        prefix $\leftarrow c$
    end if
end while
if *prefix* $\neq \Lambda$ **then**
    Output index of *prefix* in D
    Generate a new permutation π with G and apply it to D
end if
Output index of EOF in D

However, to achieve strong security the load in the dictionary α must be negligible in λ (e.g., $\alpha \leq 2^{-\lambda}$, see Section 2.5). This implies that the size d of the dictionary must be *super-polynomial* in λ. However, as noted previously, we only need to permute a small fraction of the dictionary: that is, we only need to move the *used* entries since "moving" an empty cell to another empty cell is in essence a no-op. Moving only the used entries can be done by simply generating a new random index for each entry. If there are m entries in the dictionary and l is the length of an index, then this will take $O(ml)$ time in each iteration of the loop. When generating the indices, however, there may be collisions, i.e., a generated index refers to an entry that is already used. By the birthday paradox, the probability of such a collision is $\approx m2^{l/2}$; setting $l = \mathit{poly}(\lambda)$ makes this probability negligible. Noting that $m \leq \alpha d = \alpha 2^l$ and setting α such that $\alpha 2^l$ is polynomially-bounded, we have that Slow Squeeze compression and decompression take $O(nl\alpha d)$ time, with overwhelming probability. The efficiency of Slow Squeeze is summarized in the following lemma.

Lemma 2.2. *Consider Slow Squeeze with dictionary D of size d with maximum load α and index length $l \geq \log d$. Let $m = \alpha d$ be the maximum number of used entries in D. Then, with overwhelming probability, Slow Squeeze takes $O(nml)$ time to compress or decompress a stream of n characters.*

Algorithm 2.4 Decryption algorithm for the Slow Squeeze Cipher.

Input: Key k and sequence of indices I
Output: Output character stream O or $\perp$

Initialize dictionary D with all single-character strings and EOF symbol
Read the nonce ℓ from I
Set $s \leftarrow f(k, \ell)$ and seed PRG G with s ▷ f is a PRP and G is a PRG
Set *free* to the first free entry in D
Generate a permutation π using G and apply it to D
Set $prev \leftarrow \Lambda$ and $c \leftarrow \Lambda$
Set $free \leftarrow \pi(free)$ ▷ *free* indicates the *permuted* first free position in D
while there is more input in I **do**
    Read next index i
    if $D[i]$ is defined **then**
        $c \leftarrow \text{HEAD}(D[i])$ ▷ Take the first character of $D[i]$
        $D[free] \leftarrow prev \circ c$ ▷ Insert into the *permuted* first free position
    else if $D[i]$ is not defined and $i = free$ **then**
        $c \leftarrow \text{HEAD}(prev)$
        $D[i] \leftarrow prev \circ c$
    else
        Output $\perp$ and fail ▷ Decoding failed
    end if
    if $D[i] =$ EOF and there is more input in I **then** ▷ Check for valid usage of EOF character
        Output $\perp$ and fail
    else if $D[i] =$ EOF **then**
        Exit ▷ Decoding finished
    end if
    Output $D[i]$ and set $prev \leftarrow D[i]$
    Set *free* to the first free entry in D
    Generate a new permutation π with G and apply it to D
    Set $free \leftarrow \pi(free)$
end while

2.4.2 Fast Squeeze: Efficient, Secure LZW Compression

While simple, Algorithm 2.3 is quite inefficient. After encoding a string, the entire dictionary D is randomly permuted, i.e., a permutation on $|D|$ elements must be chosen uniformly at random.[14] As stated above, the (asymptotically optimal) Fisher-Yates algorithm requires $O(n)$ time to produce a random permutation of n elements, making n calls to a source of randomness for $l = \log_2 |D|$ bits each time. In practical constructions, the source of randomness would be a PRG.[15] On each iteration of the main loop in Slow Squeeze, a single index is output ("used") and then the entire dictionary

[14] As noted above, this can be optimized so that we only permute the *non-empty* cells in the dictionary D, but we are still using many pseudorandom bits.

[15] By introducing a PRG we are biasing the distribution of permutations and limiting the number of possible permutations. For instance, $100! \approx 2^{524.8}$, requiring approximately 525 bits of internal state in the PRG. But, since we assume the PRG is secure, the distribution of permutations produced is indistinguishable from the uniform distribution over $\mathcal{P}(n)$.

must be permuted, essentially throwing away the extra pseudorandom bits produced by the PRG. That is, at each iteration, the PRG produces $|D| \log_2 |D|$ bits, but only $\log_2 |D|$ are actually used to encode data.

Speeding Things Up. To reduce this inefficiency, we re-use the "unused" randomness from the previous permutation of the dictionary. At each iteration, before outputting the index of the matched entry, we have a k out of n permutation of the dictionary entries, where k is the number of used entries. After outputting the index of an entry e, we have revealed part of the permutation and are left with a $k-1$ out of $n-1$ permutation.[16] To restore the k out of n permutation we can simply swap e with another entry chosen at random (including empty entries). See Lemma 2.4 in Section 2.5 for a proof that this is sufficient. When adding an element to the dictionary, to build a $k+1$ out of n permutation, we only need to select one of the *empty* entries at random and use it to store the new entry. The other entries can be left in place. We show in Lemma 2.5 in Section 2.5 that this is sufficient to create a random $k+1$ out of n permutation of the dictionary entries.

Malleability. Unfortunately, the ciphertexts produced by Fast Squeeze are malleable, i.e., the adversary $\mathcal{A}$ can manipulate a ciphertext to produce another different, yet valid, ciphertext. For example, $\mathcal{A}$ could simply remove the tail of a ciphertext and it would decrypt without trouble. Even if we add a special EOF marker to detect truncation attacks, the adversary can still *swap* two adjacent indices and escape detection, with high probability.[17] There is no way to prevent such an attack without creating a feed-back loop where the current plaintext symbol or output index affects the encryption of the subsequent plaintext in a non-trivial way. The cleanest solution to this is to simply use a secure MAC computed over the output indices to detect any manipulation by the adversary. This also eliminates the need for a special EOF marker and, more importantly, allows us to have a dictionary load α much closer to 1. This is in contrast to Slow Squeeze which must have α be negligible in λ to achieve strong security, ensuring that "compression" is largely asymptotic instead of practical. Using a secure MAC allows for a profoundly better compression ratio in addition to using fewer pseudorandom bits. The encryption algorithm is diagrammed in Figure 2.3 and is detailed in Algorithm 2.5.

[16] We assume that the adversary has full knowledge of the input message.

[17] This attack would fail if the consecutive indices fall in the decoding special case, where the input consists of $c \circ p \circ c \circ p \circ c$. In this situation, the second index will refer to an empty entry in the previous iteration and will cause a decoding error if used in that iteration.

Note that this scheme (i.e., a cipher plus a MAC) is the construction used in many authenticated encryption schemes. If we use an incremental MAC, such as HMAC, the MAC computation can be performed online while we are encrypting the plaintext. Most authenticated symmetric encryption schemes achieve security in one or two passes over the data, adding compression into the mix then adds another pass over the data. Thus, Fast Squeeze is no slower (in terms of passes over the data) than the best authenticated encryption schemes, and Slow Squeeze achieves the same properties with just a single pass. Note that if we omit the MAC, then our scheme provides as much security as simply composing compression and encryption in the naïve way. But, we achieve compression and encryption with a single pass while the naïve construction requires two.

Decryption. In Algorithm 2.7 we have the decryption algorithm specified for Fast Squeeze. It behaves much as the original decompression algorithm in Algorithm 2.2. Note that at the end, we verify that the computed MAC is equal to the sent MAC and fail if not. There is one subtle complication to Fast Squeeze's decryption algorithm. Since we are randomly inserting entries into the dictionary, we have to be more careful with the special case described earlier, where $prev = c \circ w$ is sent followed by $prev \circ c = c \circ w \circ c$.[18] Here, instead of testing that the received index i is equal to the free index, we pre-compute at the pseudorandom index for the next inserted entry (denoted r) and check if $i = r$. Note that this is *not* simply a matter of looking at the next few pseudorandom bits, rather it requires that we simulate the RandomInsert procedure (but without modifying the dictionary). In the main `if-else` block of Algorithm 2.7, we check if $prev \neq \Lambda$ to ensure that we skip these blocks on the first iteration through the loop. If we do not, then the character c will be mistakenly inserted into the dictionary a second time since on the first iteration $prev \circ c = c$.

Efficiency. LZW compression is efficient, taking $O(n)$ time to compress a stream of n characters—with a dictionary that performs $O(1)$ time look-ups, LZW performs just a constant amount of work at each step. For Fast Squeeze, the compression and decompression functions can utilize these efficient data structures for look-ups and processing the input/output characters as well. The essential difference between Fast Squeeze and LZW is the dictionary management, where Fast Squeeze partially randomizes the dictionary at each step with RandomSwap and RandomInsert.

[18] Recall that the difficulty we have here is that, technically, we receive an index for an entry that is not yet defined. Since this only happens in a specific case, we can resolve the ambiguity.

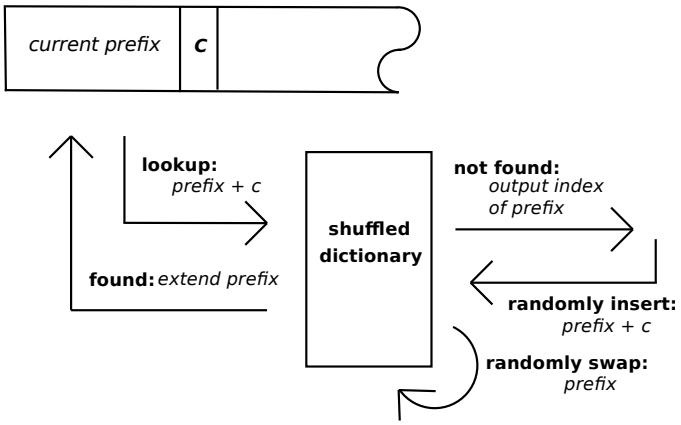


Figure 2.3: Diagram of the Fast Squeeze secure compression algorithm.

Assuming that it takes $O(l)$ time for the PRG to produce a random index of l bits, the RandomSwap step only takes $O(l)$ time. However, RandomInsert could take more. Sampling the index of an empty cell at random can be accomplished in constant time by storing those indices in a randomly permuted array. Just taking the first entry will then give a sample that is uniformly distributed; subsequent indices sampled this way would also be uniformly distributed. However, a difficulty arises in the interaction between RandomInsert and RandomSwap. In particular, RandomSwap can cause an empty cell to be filled and a used cell to become empty. Thus, we must be able to add and remove entries from the set of unused indices, ideally in constant time. But, it is not clear how to create and implement a data structure that allows for $O(1)$ random sampling in addition to $O(1)$ addition and removal.

The most straightforward way to allow efficient random sampling of empty cells is to bound the maximum load in the dictionary by some β, where $0 < \beta < 1$, and then sample dictionary indices at random (taking $O(l)$ time each for each sample) until an empty cell is found.[19] This is the approach used both in the definition of RandomInsert in Algorithm 2.6 and in our implementation. Since the load in the dictionary is $\alpha \leq \beta$, and each iteration of the while-loop is independent of the others, the number of iterations of the loop is a geometrically distributed random variable with mean $1/(1-\beta)$. Moreover, with high-probability there will be at most $\log_{1/\beta} d$ iterations, where d is the size of the dictionary (note that $d < n$, where n is the input size). Thus, we have that both encryption and decryption take $O(n(l + l \log_{1/\beta} d))$ time with high-probability.

[19] We also note that the parameter β gives a trade-off between compression/decompression speed and compression ratio: larger values of β give better compression while small β give better speed.

Algorithm 2.5 The Fast Squeeze cipher compression algorithm.

Input: keys k_{prp} and k_{mac}, nonce ℓ, input character stream I
Output: encrypted output stream O

Set $\mathit{prefix} \leftarrow \Lambda$, $c \leftarrow \Lambda$
Create PRG G and empty dictionary D
InitSqueeze(k_{prp}, k_{mac}, ℓ, G, D) ▷ Initialize PRG G, D, and MAC: see Algorithm 2.6
Output nonce ℓ
MacUpdate(k_{mac}, ℓ) ▷ Include the nonce in the MAC
while there is more input in I **do** ▷ Start compressing and encrypting stream I
    Read next character c
    if $\mathit{prefix} \circ c$ in D **then**
        $\mathit{prefix} \leftarrow \mathit{prefix} \circ c$
    else
        Output index i of prefix in D
        MacUpdate(k_{mac}, i) ▷ Update the MAC to include i
        RandomSwap(i, D, G) ▷ Swap prefix with a random entry in D
        RandomInsert($\mathit{prefix} \circ c$, D, G) ▷ Insert $\mathit{prefix} \circ c$ randomly into D
        $\mathit{prefix} \leftarrow c$
    end if
end while
if $\mathit{prefix} \neq \Lambda$ **then** ▷ Make sure we get any straggling input
    Output index i of prefix in D
    MacUpdate(k_{mac}, i)
end if
Set $m \leftarrow$ MacFinalize(k_{mac}) ▷ Finish computing the MAC
Output m

Lemma 2.3. *Consider Fast Squeeze with a dictionary D with maximum load α, size d and index length $l \geq \log_2 d$. Then, with high-probability, Fast Squeeze takes $O(n(l + l \log_{1/\beta} d))$ time to compress or decompress a stream of n characters.*

Optimality of Fast Squeeze. There are several ways to permute the dictionary of LZW to achieve security. First, the approach we take is to permute the entire dictionary with only a subset of the entries occupied at any given time. An alternative would be to permute only the used entries, e.g., if the first 100 entries of the dictionary are used, then they would be randomly permuted *among themselves*. Another alternative would be to have some, possibly dynamically growing, subset of the dictionary that is permuted at each step. Though, the chosen subset must include *all* used entries, otherwise an entry may be reused without re-randomization inducing correlations among the output indices and undermining security.

In each of these cases, though, the final position of each dictionary entry is a function of the randomness used to generate the permutation. Using true randomness ensures that the cipher has

Algorithm 2.6 InitSqueeze, RandomInsert, and RandomSwap functions.

```
1: function InitSqueeze(k_prp, k_mac, ℓ, G, D)
2:     Compute seed = f_{k_prp}(ℓ)    ▷ f is a PRP
3:     Initialize MAC algorithm with k_mac
4:     Initialize G with seed
5:     Initialize dictionary D
6:     for each single character string s do
7:         RandomInsert(s, D, G)
8:     end for
9: end function
```

```
1: function RandomInsert(s, D, G)
2:     repeat
3:         Sample a new index r using PRG G
4:     until D[r] = Λ      ▷ Until D[r] is empty
5:     D[r] ← s
6: end function
```

```
1: function RandomSwap(i, D, G)
2:     Sample a new index r using PRG G
3:     Swap D[i] and D[r]
4: end function
```

perfect, entropy-restricted secrecy (see Section 2.5, Theorem 2.1), and to achieve ER-CPA security, we must ensure that the algorithm performs only negligibly different when given pseudorandom bits. That is, we want that the advantage (over 1/2) in distinguishing the two challenge messages is negligible; moreover, we want this guarantee to be *independent* of the implementation of the permutation generation. By definition, this requires that we use a cryptographically-strong pseudorandom generator.

For Fast Squeeze, the security of the scheme lies in the k out of n random permutation of the k used dictionary entries (with n total entries). When Fast Squeeze outputs an index it partially reveals that permutation, giving a $k-1$ out of $n-1$ permutation. We show in Section 2.5, Lemma 2.4, that the routine RandomSwap restores the k out of n permutation. In addition, in Lemma 2.5, we show RandomInsert extends the permutation to a random $k+1$ out of n permutation. By examination of the proofs for Lemmas 2.4 and 2.5 we see that the functions RandomSwap and RandomInsert are both mathematically necessary and optimal (i.e., use a minimum amount of pseudorandom bits) to restore the permutation.[20] Thus, we have that Fast Squeeze is (essentially) optimal with respect to permutation-based secure LZW ciphers.

2.4.3 A Few Implementation Details

Dictionary Management. Earlier, we mentioned a technique for compressing the dictionary where each entry for a prefix p in the dictionary consists of the last character c of p and a pointer to the prefix p' (where $p' \circ c = p$). This optimization is compatible with Fast Squeeze and can greatly reduce the memory used by the dictionary.

[20] The efficiency of RandomInsert as given in Algorithm 2.6 could be improved given a data structure that can achieve all of the following: (1) $O(1)$ random sampling, (2) $O(1)$ removal from a random position, and (3) $O(1)$ random insertion. Currently, we know of no data structure that achieves all three simultaneously.

Algorithm 2.7 The Fast Squeeze decryption algorithm.

Input: keys k_{prp} and k_{mac}, input stream of indices I
Output: output character stream O

Read nonce ℓ from I
Set $\mathit{prefix} \leftarrow \Lambda$, $c \leftarrow \Lambda$
Create PRG G and empty dictionary D
InitSqueeze(k_{prp}, k_{mac}, ℓ, G, D) ▷ Initialize PRG G, D, and MAC: see Algorithm 2.6
MacUpdate(k_{mac}, ℓ)
while there is another index in I **do**
    Read next index i
    MacUpdate(k_{mac}, i)
    Set $r \leftarrow$ IndexNextInsertion(D, G) ▷ Get the index r of the new entry $\mathit{prev} \circ c$
    if $D[i]$ is undefined and ($i \neq r$ or $\mathit{prev} = \Lambda$) **then**
        Output $\perp$ and fail ▷ Decoding failed: invalid index
    end if
    if $D[i]$ is defined and $i \neq r$ and $\mathit{prev} \neq \Lambda$ **then** ▷ i is valid and not part of a collision
        $c \leftarrow$ head($D[i]$)
        RandomInsert($\mathit{prev} \circ c$, D, G)
    else if $i = r$ and $\mathit{prev} \neq \Lambda$ **then** ▷ Special case: input was $c \circ w \circ c \circ w \circ c$
        $c \leftarrow$ head(prev)
        RandomInsert($\mathit{prev} \circ c$, D, G)
    end if
    Output $D[i]$
    $\mathit{prev} \leftarrow D[i]$
    RandomSwap(i, D, G) ▷ Move prev to a random position
end while
Set $m \leftarrow$ MacFinalize(k_{mac}) ▷ Finish computing the MAC
Read received MAC m'
if $m \neq m'$ **then**
    Output $\perp$ and fail
end if

If the load α of the dictionary is very small, then it becomes inefficient to store the entire dictionary since it is so sparse. In this situation the dictionary would be stored *implicitly* with some intermediate data structure storing the entries, where each entry contains a pointer to the prefix for that entry, the "extending" character, and the entry's index in the implicit dictionary. This can be efficiently implemented using a hash table or a trie; the latter is used in our implementations of both Slow and Fast Squeeze.

Management of the dictionary as it grows is an important part of implementing both LZW compression and Fast Squeeze. Fortunately, all of the dictionary management techniques used in LZW compression are also compatible with Fast Squeeze, but some care is needed. For example, with a dynamically growing dictionary in LZW, when the dictionary becomes filled, we can double it in size and increase the length of all of the indices by one bit. But, increasing the length of a

Algorithm 2.8 IndexNextInsertion simulates RandomInsert without modifying D.

```
1: function IndexNextInsertion(D, G)                    ▷ Simulate RandomInsert
2:     Set i ← 0
3:     repeat
4:         Sample a new index r using PRG G
5:         i ← i + 1
6:     until D[r] = Λ                                   ▷ Until D[r] is empty
7:     Rewind G by i samples
8:     return r
9: end function
```

random index is a little trickier. A naïve approach is to generate a random bit for each index and then prepend or append it, or even insert it at a random position in the index. But, this does *not* guarantee a uniform distribution of indices. In particular, since all of the original (say l-bit) indices are distinct, then whether prepending or appending it is impossible for two of the $l + 1$ bit indices to be different in only the prepended/appended bit (similarly for inserting a bit). To ensure that indices are uniformly distributed, the dictionary must be randomly permuted whenever increasing its size.[21]

As an alternative to keeping every entry in the dictionary and growing dynamically, we can use a deterministic eviction policy invoked when the dictionary becomes full. This uses of the pseudorandomness more efficiently since the eviction itself requires no randomness and the insertion of the new entry requires the usual amount. An example eviction policy could be the standard least-recently-used or least-frequently-used policies. Other eviction policies are possible. Any eviction policies that depend on the *ordering* of entries in the dictionary are incompatible with Fast Squeeze since the layout of the dictionary is randomized.

Memory Requirements. The LZW algorithm lends itself to compact in-memory representations and Fast Squeeze inherits this property. We implemented the dictionary using a trie, with all possible byte values as the input alphabet, and analyze its memory usage here, focusing first on the memory footprint of plain LZW and then analyzing Fast Squeeze. To avoid reserving space at each node for pointers to all possible children (i.e., 256 pointers), we stored the child pointers in dynamically allocated slabs of 64 pointers. We use bitmasks to indicate the presence or absence of a child and thereby avoid the cost of initializing the slabs of pointers. This combination decreased memory requirements allowing for better cache utilization and increased the speed of the algorithm. But there are more memory-efficient implementations. One such implementation is to store the children in a

[21] Since the permutation can be done in $O(ln)$ time, where n is the dictionary size and l is the index length, permuting the dictionary when growing has an amortized overhead of $O(l)$ at each step of encoding.

linked list so that each node only needs to store a single pointer for all of its children. We implemented just such a scheme and analyze its speed in Section 2.6.6; here, we section focus on its memory requirements. Specifically, we analyze the memory used when the linked list is implemented as an intrusive doubly-linked list. An intrusive doubly-linked list gives us an $O(1)$ move-to-front operation which allows us to store the children in most-recently-used-first order, giving a significant speed-up.

This compact representation requires 1 byte for the character corresponding to that node, $b = \lceil l/8 \rceil$ bytes for the index of length l, a pointer to the beginning of the list and the 2 pointers for the intrusive list. This requires $1 + b + 3p$ bytes, where p is the number of bytes for a pointer. In our implementation $b = 2$ and $p = 4$ or 8, depending on the architecture. With $p = 8$, as on the 64-bit Core 2 Quad processor, each trie node requires 27 bytes. With $p = 4$, as on the ARM11 processor, each trie node is 15 bytes in size. With 12-bit indices, this gives a total dictionary size of $2^{12} * 27 = 108$KB on the Core 2 Quad, and $2^{12} * 15 = 60$KB on the ARM11. Note that both of these are smaller than the default 256KB and 300KB of memory used by gzip and bzip2, respectively. Fast Squeeze also requires the compressor to maintain a mapping of indices to trie nodes to efficiently perform the RandomInsert routine (i.e., to quickly determine whether or not a given dictionary index has been allocated). This adds an additional $2^l * p$ bytes, which, continuing our example, would be an additional $2^{12} * 8 = 32$KB on a 64-bit architecture and $2^{12} * 4 = 16$KB on a 32-bit architecture. Note that the totals of 140KB and 76KB are still less than the memory requirements of gzip and bzip2.

For decompression, we must proceed bottom-up through the trie to recover the input character by character instead of top-down. Because of this, each node will need a pointer to its parent, but it will *not* need to keep a list of children. Overall, each trie node must store its corresponding character and a pointer to its parent for a total of $1 + p$ bytes per node. As a speed optimization, while decompressing each node can save a pointer into the buffer of decompressed data corresponding to the beginning of the string for that node (as well as save the length of the string). This adds another $p + b$ bytes to the node.[22] This avoids traversing all the way to the root each time a node in the dictionary is used. This optimization gives a total of $1 + 2p + b$ bytes per node. On a 64-bit architecture and 12-bit indices, this is 19 bytes; on a 32-bit architecture this is 11 bytes.

Just like the encoder, the decoder must also maintain a mapping of indices to trie nodes to avoid searching the entire trie for a specific node. This gives total memory requirements for decoding to be

[22] The string represented by a trie node can never be more than 2^l bytes long since that would require more nodes than can fit in the dictionary.

$2^{12} * 8 + 2^{12} * 19 = 108$KB on a 64-bit machine and $2^{12} * 4 + 2^{12} * 11 = 60$KB on a 32-bit machine. Decompression with gzip, however, requires just 44KB on a 32-bit machine. However, without the optimization of keeping pointers into the decompressed buffer, LZW only requires 9 and 5 bytes, giving total of 68KB and 36KB.

The above space analyses are for LZW in general rather than Fast Squeeze. In our implementation of Fast Squeeze, we bound the load in the dictionary by 1/2, but we still need the full inverse mapping of indices to trie nodes. For decoding, if we include the string-length optimization, then decompression requires $2^{12} * 8 + 2^{11} * 19 = 70$KB and $2^{12} * 4 + 2^{11} * 11 = 38$KB on 64 and 32-bit architectures, respectively. Without the string-length optimization, squeeze decompression requires $2^{12} * 8 + 2^{11} * 9 = 50$KB $2^{12} * 4 + 2^{11} * 5 = 26$KB. Note that in both with and without the optimization, on a 32-bit machine, Fast Squeeze requires less memory than gzip.

2.5 Security Analyses

In this section we prove the (concrete) ER-CCA-2 security of both of the Slow and Fast Squeeze ciphers. We proceed by first proving that each is ER-CPA secure and then proving that each has the property of existential ciphertext unforgeability (defined in Section 2.2.4), which implies ER-CCA-2 security. (Recall that the original paper [78] showed that IND-CPA security combined with ciphertext unforgeability implies IND-CCA-2 security. Since the proof was independent of plaintext length, the same implication holds for ER-CPA and ER-CCA-2 security.)

2.5.1 ER-CPA Security

The proof of ER-CPA security below applies to both constructions, provided that at each step the layout of the dictionary is a random permutation of the entries. For Slow Squeeze, this is inherent in the construction; for Fast Squeeze we will need to do a bit more work. We proceed by first proving two lemmas that establish two loop-invariants for Fast Squeeze encryption and decryption. Then, we will prove the theorem that both Fast and Slow Squeeze are ER-CPA secure.

Intuitively, if the entries of the dictionary are randomly permuted, then an adversary observing the output cannot derive any correlations between the output indices and the input strings. In Fast Squeeze, each index output provides a glimpse into the internal ordering of the entries in the

dictionary.[23] To counteract this, RandomSwap seeks to restore the random permutation of the dictionary entries and RandomInsert works to extend the permutation to include a new entry. We refer to the mapping $\pi(a) = b$ to be an "entry" in the permutation π.

Lemma 2.4. *Given a random permutation of k out of n elements, if we reveal any entry, applying* RandomSwap *to that entry produces a random permutation of k out of n elements.*

Proof. Given a random k out of n permutation, there are $n!/(n-k)!$ possible such permutations. After revealing one entry, we have a $k-1$ out of $n-1$ permutation over the remaining entries, giving $\frac{(n-1)!}{((n-1)-(k-1))!} = \frac{(n-1)!}{(n-k)!}$ possibilities. RandomSwap selects an element at random from the n possible positions (including empty positions) and swaps it with the revealed element. Each of the n possible swaps produces a unique configuration, giving a total of $n\frac{(n-1)!}{(n-k)!} = \frac{n!}{(n-k)!}$ possible permutations—i.e., a k out of n permutation. Note that the $k-1$ out of $n-1$ permutation is uniformly distributed, and since the swap is also uniformly distributed, the final k out of n permutation is also uniformly distributed. □

Thus, RandomSwap restores the random k out of n permutation after each invocation. As a corollary, this ensures that if we stop inserting elements into the dictionary (i.e., when it is full) but keep calling RandomSwap at the appropriate time, then we will maintain the security of the scheme. Now we prove that adding an element to the dictionary through RandomInsert produces a random permutation on the entries.

Lemma 2.5. *Given a random k permutation of n elements,* RandomInsert *produces a random $k+1$ out of n permutation of the elements.*

Proof. Note that we are given a k out of n permutation that is uniformly distributed. RandomInsert selects a position at random from the $n-k$ remaining positions and inserts the new entry there. This gives a total of $(n-k)\frac{n!}{(n-k)!} = \frac{n!}{(n-k+1)!}$ possibilities, which is exactly the number of $k+1$ permutations of n elements. Note that since the original permutation was uniformly distributed, as was our selection of the new position, the resulting permutation is also uniformly distributed. □

Thus, we have the after each iteration of the Fast Squeeze encoder or decoder, the positions of the dictionary's entries form a random k out of n permutation.

[23] For Slow Squeeze, this leakage is immaterial.

Now we prove that both the Fast and Slow Squeeze ciphers are ER-CPA secure. Both constructions are, at the most basic level, a PRG integrated into the LZW compression algorithm with a PRP for session key derivation. This simple combination gives us straightforward proof of security whereby we can reduce ER-CPA security to the indistinguishability of the PRG and the PRP. In the proof, for simplicity, we assume all encryption oracle queries take time at most t_e.[24] Let $\mathcal{C}(G, f)$ denote either Fast or Slow Squeeze instantiated with PRG G and PRP f.

Theorem 2.1 (Slow/Fast Squeeze ER-CPA Security)**.** *Let G be a PRG that is (t_G, r, ε_G)-reseedably-indistinguishable that takes seeds of length λ, and let f be a (t_f, r, ε_f)-indistinguishable PRP over $\{0,1\}^\lambda$. Then the (Fast/Slow) Squeeze cipher $\mathcal{C}(G, f)$ is $(t', r-1, \delta)$-ER-CPA secure where $t' = \min\{t_G - rt_e, t_f - rt_e\}$ and $\delta = 2(\varepsilon_G + \varepsilon_f) + \frac{r(r-1)}{2^{\lambda+1}}$.*

Proof. Suppose $\mathcal{A}$ can break the ER-CPA security of (Slow/Fast) Squeeze and that f is a truly random *function*. Consider the following distinguisher $\mathcal{D}$ for G using $\mathcal{A}$ as a subroutine with access to a PRG oracle $\mathcal{O}$. On input 1^λ, $\mathcal{D}$ runs $\mathcal{A}$ and for each query message m from $\mathcal{A}$, $\mathcal{D}$ simply requests a re-seeding of $\mathcal{O}$, generates a unique nonce ℓ, and then encrypts m following the specification of $\mathcal{C}$. When given the challenge messages m_0 and m_1, $\mathcal{D}$ selects a random bit b, encrypts m_b to produce c_b and gives c_b to $\mathcal{A}$. When $\mathcal{A}$ outputs its guess b', the distinguisher $\mathcal{D}$ outputs 1 if $b' = b$, and 0 otherwise.

First note that since $\mathcal{A}$ runs in polynomial time, so does our distinguisher $\mathcal{D}$. In analyzing the success probability of $\mathcal{D}$, we must analyze the result of feeding truly random bits to $\mathcal{A}$. We know that, if the input is pseudorandom, $\mathcal{A}$ will succeed with advantage ε_G over random. Secondly, note that we do not use an initialization vector for generating the seed for the PRG in $\mathcal{O}$. However, since for each seeding of the PRG, the seed is chosen uniformly at random, this is exactly equivalent to using a random function f applied to a unique ℓ for each encryption query (i.e., the distributions of seeds are identical). We will later remove the assumption that f is a random function. Now, if the bits given to $\mathcal{A}$ are random, then the possible encryptions of m_0 and m_1 are *identically* and *uniformly* distributed, so $\mathcal{A}$'s success probability is exactly $\frac{1}{2}$. If the bits are pseudorandom, then $\mathcal{A}$ succeeds with some advantage δ, giving $\mathcal{D}$ a success probability of $\frac{1}{2} + \frac{\delta}{2}$. Since G is (t, r, ε_G)-indistinguishable, we have that $\delta \leq 2\varepsilon_G$, and it must be that there were at most $q = r - 1$ messages queried by $\mathcal{A}$, and the running time of $\mathcal{A}$ is $t' = t_G - qt_e$.

[24] Normally, one assumes $O(1)$ time for oracle queries, but we allow for highly variable input lengths, giving a (possibly) wide variance in computation time. Hence, we include an explicit bound on the encryption time.

RF to PRP. We remove the assumption that f is a random function (RF) by first replacing f with a random permutation (RP). Note that for any distinguisher $\widetilde{\mathcal{D}}$, the only way to distinguish an RP from an RF is to find a collision in the latter. Thus, $\widetilde{\mathcal{D}}$ has an advantage of at most $q(q-1)/2^{\lambda+1}$, where q is the number of oracles queries and λ is the function output size. So the advantage gained by any adversary against $\mathcal{C}$ is at most $r(r-1)/2^{\lambda+1}$, where r is the number of reseeding requests. Now, replace f with a PRP and consider the following $\mathcal{F}$ given access to an oracle $\mathcal{O}_f$ that may be an RP or a PRP. If $\mathcal{F}$ believes $\mathcal{O}_f$ to be a pseudorandom, then it outputs 1, otherwise it outputs 0. $\mathcal{F}$ runs $\mathcal{A}$, and for each query m, $\mathcal{F}$ generates a fresh nonce ℓ, queries $\mathcal{O}_f(\ell) = s$, and then encrypts m in the normal way using the PRG G seeded with s. This process repeats for the subsequent queries and the challenge messages. Eventually, $\mathcal{A}$ outputs a guess b'; if $b' = b$, $\mathcal{F}$ outputs 1, else it outputs 0.

If $\mathcal{O}_f$ is a random permutation, then $\mathcal{A}$'s success probability is at most $\frac{1}{2} + r(r-1)/2^{\lambda+1} + 2\varepsilon_G$, which means that $\mathcal{F}$ is *incorrect* with the same probability. If $\mathcal{O}_f$ is a PRP, then $\mathcal{A}$'s chance of success is at most δ greater than $\frac{1}{2} + r(r-1)/2^{\lambda+1} + 2\varepsilon_G$, for some δ. This gives a total success probability of $\frac{1}{2}(\frac{1}{2} - r(r-1)/2^{\lambda+1} - 2\varepsilon_G) + \frac{1}{2}(\frac{1}{2} + r(r-1)/2^{\lambda+1} + 2\varepsilon_G + \delta) = \frac{1}{2} - \varepsilon_G + \varepsilon_G + \frac{\delta}{2} = \frac{1}{2} + \frac{\delta}{2}$. Since f is a (t_f, r, ε_f)-indistinguishable, we know that $\delta/2 \leq \varepsilon_f$. This implies that $\mathcal{A}$'s advantage in distinguishing m_0 from m_1, when G is a PRG and f is a PRP, is $2(\varepsilon_G + \varepsilon_f) + r(r-1)/2^{\lambda+1}$.

Note that the number of encryption oracle queries is exactly equal to the number of reseedings of G. So, we have that the number of oracle queries for f is upper-bounded by r. Finally, we have that $\mathcal{F}$'s running time is upper-bounded by t_f, so $\mathcal{A}$'s running time is upper-bounded by $t_f - rt_e$. □

Corollary 2.1. *Let G be a computationally secure, reseedably-indistinguishable PRG that takes seeds of length λ, and let f be a computationally secure PRP over $\{0,1\}^\lambda$, then the (Slow/Fast) Squeeze cipher $\mathcal{C}(G, f)$ is ER-CPA secure.*

2.5.2 Ciphertext Unforgeability

In this section, we will prove that both Slow and Fast Squeeze possess ciphertext unforgeability—a property first defined in [78], see Definition 2.11 in Section 2.2.4. We will prove this for Slow Squeeze first. For Fast Squeeze, the result follows simply from the unforgeability of the MAC. Recall that a cipher C is said to be (t, q, ε)-unforgeable if for all PPT $\mathcal{A}$ running in time t making at most q encryption oracles succeeds in producing *any* valid ciphertext with probability at most ε.

Slow Squeeze Unforgeability

Intuitively, the proof of unforgeability for Slow Squeeze works by setting the load in the dictionary α to be a value that is negligible in the security parameter λ. With the dictionary so sparsely populated, guesses for a particular entry's index and manipulations of a valid ciphertext will fail with overwhelming probability. Hence, the security of Slow Squeeze directly reduces to the quality of the randomness produced by the PRG, which we show below. As before, let t_e denote the time to encrypt one ciphertext and let $\mathcal{C}_\alpha(G, f)$ denote Slow Squeeze instantiated with PRG G, PRF f, and maximum load α. Let l be the length of the dictionary indices in bits (i.e., the dictionary has 2^l entries), but only an α fraction are used. Let $\mathcal{O}$ denote the encryption oracle.

Theorem 2.2 (Slow Squeeze Ciphertext Unforgeability)**.** *Let $\mathcal{C}_\alpha(G, f)$ be the Slow Squeeze cipher with load at most α, where G is a (t_G, r, ε_G)-reseedably-indistinguishable PRG and f is a (t_f, r, ε_f)-indistinguishable PRP. Then, $\mathcal{C}_\alpha(G, f)$ is (t', q, ε)-unforgeable (i.e., existentially unforgeable) for $\varepsilon = \alpha + 2(\varepsilon_G + \varepsilon_f)$, $q = r$, and $t' = \min\{t_G - qt_e, t_f - qt_e\}$.*

Proof. First, assume that G, used by $\mathcal{O}$ for encryption, produces truly random bits. To use true randomness consistently with the format of the encryption, we let the nonce ℓ be the index of a row in a table of random bits generated lazily by G (so that the table is at most polynomial in size). We run $\mathcal{A}$ on input 1^λ and let c be the attempted forgery. The nonce ℓ used in c may or may not correspond to the nonce for a message previously queried to the encryption oracle. If it does not, then it corresponds to a fresh row of randomness in the table. If it does, then $\mathcal{A}$ has some partial information about the layout of the dictionary that may aid in generating c. We will address each case separately.

Case: nonce ℓ is fresh. If ℓ is fresh, then the dictionary D was permuted with fresh randomness and the entries are all uniformly distributed. Recall that in Slow Squeeze each encrypted message must end with the EOF marker. Since there is exactly one valid index for this symbol, regardless of what $\mathcal{A}$ does with the rest of the ciphertext, $\mathcal{A}$ correctly ends the message with probability exactly $\frac{1}{2^l}$. Thus, in this case, $\mathcal{A}$ succeeds with probability at most $\frac{1}{2^l} < \alpha$.

Case: nonce ℓ is reused. First note that if ℓ is reused, there is some overlap in randomness used in the forged ciphertext c and a previous query q. (Note that this overlap may consist of only the nonce.) Ignoring the shared ℓ, consider c and q as finite sequences of indices, $c_1, \ldots, c_a$ and $q_1, \ldots, q_b$

respectively. Note that there must at least one place where c and q differ, i.e., there exists an index $i \leq \min(a, b)$ such that $c_j = q_j$ for $1 \leq j \leq i-1$ but $c_i \neq q_i$.

Suppose there is no such i, then either c is a proper prefix of q or vice versa. If c is a proper prefix of q, then decryption of c will fail since the decrypted message is missing the EOF symbol. If q is a proper prefix of c, then c has extra indices *after* the EOF, which will also cause a decryption failure.

So, consider i such that $c_i \neq q_i$, and let $n \leq \alpha 2^l$ be the number of valid entries in the dictionary. The value c_i is an index in a randomly permuted dictionary where only an α-fraction of indices are valid. But, $\mathcal{A}$ already knows the value of one index, specifically q_i. Since $c_i \neq q_i$, $\mathcal{A}$ is seeking to find one of the $n-1$ valid indices randomly distributed in a set of $2^l - 1$ possible indices. Thus, $\mathcal{A}$ succeeds with probability $\frac{n-1}{2^l-1} < \frac{n}{2^l} \leq \alpha$.

Moving to pseudorandomness. To remove the assumption of using truly random bits, we provide a straightforward reduction of the unforgeability of the cipher to the indistinguishability of the PRG G. Recall that our PRG G is (t_G, r, ε_G)-reseedably-indistinguishable and assume that we have a ciphertext forger $\mathcal{A}$ who runs in time t', makes q queries, and succeeds with probability δ. We build our distinguisher $\mathcal{D}$ to attack G (i.e., distinguish its output from a random string) by simply requesting a reseeding the oracle $\mathcal{O}$ whenever $\mathcal{A}$ makes an encryption query. We then use the returned string as the source of randomness for encrypting the query. If $\mathcal{A}$ succeeds in producing a forgery, we output 1, else we output 0. We assume that $q \leq r$. If $q > r$, then for the $(r+1)$-th to q-th queries, we can just use random bits for the encryption. If this changes the success of $\mathcal{A}$ by more than a negligible amount, then we will have created a distinguisher for G which makes at most r reseeding requests. So, without loss of generality, $q \leq r$.

If the strings returned by $\mathcal{O}$ to $\mathcal{D}$ are random, then $\mathcal{A}$'s probability of success is at most α. Note that if $\mathcal{A}$ succeeds in this case, then $\mathcal{D}$ outputs an *incorrect* guess. If the input strings are pseudorandom, then $\mathcal{A}$'s probability of success is at most $\alpha + \epsilon$ for some ϵ. This gives $\mathcal{D}$ a probability of success of $\frac{1}{2}(1-\alpha) + \frac{1}{2}(\alpha + \epsilon) = \frac{1}{2} + \epsilon/2$. Since we know that G is (t_G, r, ε_G)-reseedably-indistinguishable, we have that $\epsilon \leq 2\varepsilon_G$ and $t' = t_G - qt_e$. Finally, note that the reduction relating the security of Slow Squeeze to the security of the PRP f proceeds exactly as in the above proof of ER-CPA security. Thus we have that $t' = \min\{t_G - qt_e, t_f - qt_e\}$ and $\varepsilon = \alpha + 2(\varepsilon_G + \varepsilon_f)$. □

Combining this with Lemma 2.1, we have that Slow Squeeze is ER-CCA-2 secure, if α is negligible in λ. This result is detailed in the following corollary, which is an adaption of Theorem 1 from [78].

Corollary 2.2. *Let $\mathcal{C}_\alpha(G, f)$ be the Slow Squeeze cipher with load at most α, using a (t_G, r, ε_G)-reseedably-indistinguishable pseudorandom generator G and a (t_f, r, ε_f)-indistinguishable pseudorandom permutation f. Then, $\mathcal{C}_\alpha(G, f)$ is $(t', q_e, q_d, \varepsilon)$-ER-CCA-2 secure for $t' = \min\{t_G - rt_e, t_f - rt_e\}$, $q_e = r - 1$, and $\varepsilon = q_d\delta_1 + \delta_2$, where $\delta_1 = 2(\varepsilon_G + \varepsilon_f) + \frac{r(r-1)}{2^{\lambda+1}}$ and $\delta_2 = \alpha + 2(\varepsilon_G + \varepsilon_f)$.*

Corollary 2.3. *Let $\mathcal{C}_\alpha(G, f)$ be the Slow Squeeze cipher where G is a computationally secure, reseedably-indistinguishable PRG that takes seeds in $\{0,1\}^\lambda$, f is a computationally secure PRP over $\{0,1\}^\lambda$, and α is negligible in λ. Then, $\mathcal{C}_\alpha(G, f)$ is ER-CCA-2 secure.*

Fast Squeeze Unforgeability

Proving ciphertext unforgeability for Fast Squeeze is a straightforward result as the ability to produce a valid ciphertext is dependent on the ability to forge a valid MAC. Again, we have t_e denote the time to encrypt one encryption query. Let $\mathcal{C}(G, f, M)$ denote Fast Squeeze instantiated with pseudorandom generator G, pseudorandom permutation f, and MAC scheme M.

Lemma 2.6. *Let $\mathcal{C}(G, f, M)$ be the Fast Squeeze cipher where G is a PRG that is (t_G, r, ε_G)-reseedably-indistinguishable, f is a (t_f, r, ε_f)-indistinguishable PRP, and M is a (t_m, q, ε_m)-unforgeable MAC. Then, $\mathcal{C}$ is (t, q, ε)-unforgeable (i.e., existentially unforgeable) for $t = t_m - qt_e$ and $\varepsilon = \varepsilon_m$.*

Proof. Suppose we have algorithm $\mathcal{A}$ that breaks the unforgeability of $\mathcal{C}$ with probability γ. Moreover, suppose we have an oracle $\mathcal{O}$ for the MAC scheme M. We first run $\mathsf{Gen}_{\text{prp}}(1^\lambda)$ to get k_{prp} and then run $\mathcal{A}$ on input 1^λ. For each encryption query m from $\mathcal{A}$, we chooses a fresh nonce ℓ at random from $\{0,1\}^\lambda$ and then generate a session seed s as described in Algorithm 2.5 and seed G with it. We then encrypt m in the expected way to produce the ciphertext c (which includes the nonce ℓ prepended to it). We then query $\mathcal{O}$ with c to obtain the tag t and return $c \circ t$ to $\mathcal{A}$. Eventually, after making at most q queries, $\mathcal{A}$ outputs its forgery $\tilde{c} = \ell' \circ c' \circ t'$. We compute the seed s' from ℓ', seed G, and then decrypt in the usual way but without computing the MAC. If the ciphertext decrypts successfully, we then output $(\ell' \circ c', t')$ as our attempted forgery.

Since $\mathcal{A}$ succeeds when $\tilde{c}$ was *not* the result of a query to its encryption oracle, it must be that c' was not given as a query to $\mathcal{O}$ and the pair $(\ell' \circ c', t')$ is a valid MAC forgery attempt. If $\mathcal{A}$ successfully forges a ciphertext, then $\mathcal{B}$ succeeds in producing a forgery for M. Thus, we have that $\gamma \leq \varepsilon_m$ and the run time of $\mathcal{A}$ is $t \leq t_m - qt_e$. □

Note that the unforgeability is independent of the security of the PRG and the PRP used. Again, combining this with Lemma 2.1, we have that Fast Squeeze is ER-CCA-2 secure. This is detailed in the following corollaries.

Corollary 2.4. *Let $\mathcal{C}(G, f, M)$ be the Fast Squeeze cipher where G is a (t_G, r, ε_G)-reseedably-indistinguishable PRG, f is a (t_f, r, ε_f)-indistinguishable PRP f , and M is a (t_m, q, ε_m)-unforgeable MAC. Then, $\mathcal{C}$ is $(t, q_e, q_d, \varepsilon)$-ER-CCA-2 secure with $t = \min\{t_G, t_f, t_m\} - q_e t_e$, $q_e = \min\{r - 1, q\}$, and $\varepsilon = q_d\varepsilon_m + 2(\varepsilon_G + \varepsilon_f) + \frac{r(r-1)}{2^{\lambda+1}}$.*

Corollary 2.5. *Let $\mathcal{C}(G, f, M)$ be the Fast Squeeze cipher where G a computationally secure, reseedably-indistinguishable PRG that takes seeds in $\{0,1\}^\lambda$, f is a computationally secure PRP over $\{0,1\}^\lambda$, and M is an existentially-unforgeable MAC scheme. Then $\mathcal{C}(G, f, M)$ is ER-CCA-2 secure.*

Attacks on Fast & Slow Squeeze

There are several possible attacks on these schemes. The security of each depends crucially upon the uniqueness of the nonce ℓ. Reuse of a nonce will cause two (or more) ciphertexts to have the same dictionary randomization and would be vulnerable to known-plaintext attacks. Indeed, repeated use of a nonce turns Slow and Fast Squeeze into a fancy substitution ciphers. Correlations among nonces, however, are allowed since the PRP will send them to different, pseudorandom outputs. But, if the PRP is weak, correlated nonces may have correlated outputs and weaken the security of the PRG.[25]

Fast Squeeze is also vulnerable to timing attacks as input data with different levels of compressibility will take different amounts of time to compress even if the output ciphertexts are the same length. For instance, suppose there are two messages: one consisting entirely of a single, repeated character and the other consisting of random characters. Even if the strings compress to the same length, the first string will take less time to compress. This is because at each iteration of the compression loop, the index output is the index of the most recently inserted entry, which is very likely to be resident in highest level cache. The random string, however will have poor cache performance since the next entry to be matched at each iteration will be a random entry in the dictionary, and hence much less likely to be cache-resident. This timing side-channel allows an attacker to easily distinguish the encryption of these two messages. More generally, the number of iterations of the main compression loop before outputting an index is *data-dependent* and thus inherently vulnerable to timing attacks

[25] PRG security guarantees assume that the seeds are independent and random, so correlated seeds may produce weakened streams. See [75] for an analysis of this problem and constructions secure against malicious inputs.

that leak some of the plaintext. However, such data-dependent timing attacks are inherent to *all* combined compression and encryption schemes and are not unique to Fast Squeeze.

2.6 Experiments

In this section we provide a summary of experiments evaluating both the compressibility and speed of Fast Squeeze (which we will refer to simply as squeeze) as compared to standard algorithms. We compare squeeze to plain LZW as well as gzip and bzip2 since they are among the most commonly deployed compression algorithms. The test data used is the Canterbury corpus provided in [118] and described in [7]. It is a standard corpus of documents for testing lossless compression algorithms with various types of test files. We compare the algorithms on a representative subset of test files consisting of a text document of random digits, the Bible from Project Gutenberg, a segment of E.coli DNA, a snapshot of the CIA World Factbook, the first million digits of the mathematical constant π, and a C source file. These files highlight the varying performance of the Fast Squeeze cipher on different inputs.

The tests were performed primarily on a 2.4GHz, Intel Core 2 Quad processor with 4GB of RAM running Debian Linux 7.0 (wheezy). We also test the algorithms on a Raspberry Pi (Model B) running Raspbian Linux (based on Debian wheezy) on an ARM11 processor at 700 MHz with 512MB of RAM (see Section 2.6.5). We consider two possible dictionary sizes for squeeze: 2^{12} entries and 2^{16} entries—denoted squeeze-12 and squeeze-16 respectively; both use the Salsa20 stream cipher (see [15]) as the PRG and AES for the PRP. We use two different dictionary management schemes. The first scheme is the simplest where it freezes the dictionary once it is full, i.e., no more entries are added or removed. The second scheme uses a least-recently-used (LRU) eviction policy to remove entries from the dictionary when it is full. In both schemes (and with both sizes of indices), the load in the dictionary is at most 0.5.

We also compare squeeze to LZW using two combinations of encryption with LZW: (1) streaming encryption with Salsa20, encrypting each index as it is output; and (2) block encryption with AES, encrypting all of the compressed data at once. These variants are denoted lzw-aes-b and lzw-salsa-b, where $b \in \{12, 16\}$. We implemented LZW using the same data structures and optimizations (where applicable) as we used in squeeze, allowing us to better measure the performance impact of our modifications to LZW. Each of gzip and bzip2 are run at their default compression levels and the

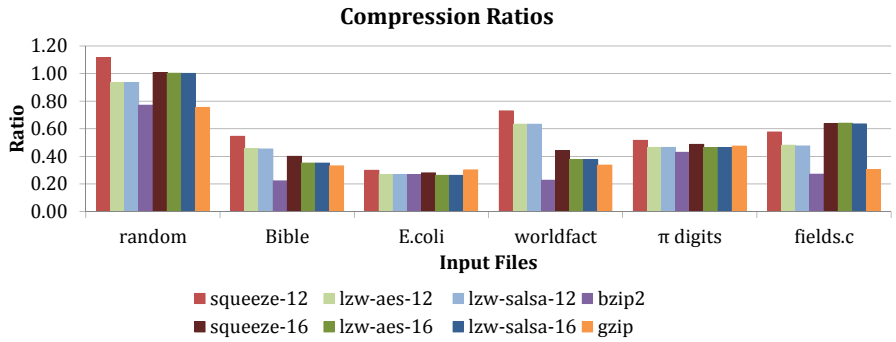


Figure 2.4: Chart comparing squeeze with 12-bit and 16-bit indices to standard compression algorithms as well as LZW combined with AES and Salsa20 encryption. Lower numbers indicate better compression.

executables were compiled with `gcc` 4.7.2 using the -O2 flag and the OpenSSL v1.0.1e library was used for AES encryption. All results are the average of 20 trials.

Summary. Overall, our experiments show that squeeze achieves compression ratios comparable to the standard algorithms LZW, gzip, and bzip2, though squeeze with 16-bit indices provides much better compression than when using 12-bit indices (see Figure 2.4, Section 2.6.1). In addition, we find that squeeze with 12-bit indices has compression speeds exceeding those of gzip and bzip2, though lagging behind the simple combination of LZW and AES/Salsa20 encryption. For decompression, squeeze has speeds comparable to gzip (and exceeding bzip2), but noticeably lower than LZW. When using 16-bit indices, both LZW and squeeze are slower than gzip (but still faster than bzip2) in both compression and decompression, but the performance difference between LZW and squeeze is much smaller (see Figure 2.5). We show similar results in Section 2.6.5 when testing on ARM11.

We compare squeeze to previous secure LZW schemes in Section 2.6.3 and show that Fast Squeeze is 2-3 *orders of magnitude* faster than previous schemes while Slow Squeeze is a modest 3% faster. In Section 2.6.4, we show that the use of least-recently-used (LRU) eviction provides better compression ratios for squeeze and LZW, and, moreover, the compression ratios are close to those of gzip and bzip2. However, the performance suffers greatly and the scheme is 4–5 times slower compared to having no evictions (compare Figures 2.5 and 2.10).

2.6.1 Compression Ratio

First, we examine the compression ratio of squeeze as compared to LZW, gzip, and bzip2. The compression ratio is defined as the (compressed) output size divided by the (uncompressed) input size with a smaller number signifying better compression. Note that, in general, a larger dictionary implies that we can achieve better compressibility. The load of the dictionary in squeeze is kept to 0.5 and once the dictionary is full, no entries are evicted or added.[26] Each of the algorithms gzip and bzip2 are run at the default compression level of 3.

We can see in Figure 2.4 that the compression ratio achieved by squeeze-12 and squeeze-16 is comparable to that of the other algorithms. On some data squeeze is quite competitive, while on other data it does relatively poorly. Part of the poor performance is due to the dictionary management. If the dictionary fills up, then the algorithm can no longer adapt to the input and will provide poor compression if the characteristics of the data change. An example where this does not matter is on the E.coli test data. The DNA of E.coli consists of only a few characters and contains many of the same patterns throughout—i.e., it all "looks" the same. However, with the C source code file, squeeze does not perform well since source code does not exhibit as much regularity as DNA or English prose. Do note that use of 16-bit indices does, indeed, given better compression in many cases. It does, however, give worse compression for small files (e.g., `fields.c`).

2.6.2 Compression Speed

In Figure 2.5, we compare the compression and decompression performance of squeeze as compared to LZW, gzip, and bzip2. Again we compare both 12-bit and 16-bit indices for squeeze and LZW, and use each of gzip and bzip2 at compression level 3 followed by encryption with AES in CBC mode. The test files include the "large" files from the Canterbury corpus (the first for groups of columns in Figure 2.5), which allow us to see how the algorithms perform at steady-state. We also include the performance of a file filled with random digits between 0 and 9, the first million digits of π, and a C source file.

Overall, the speed of squeeze with 12-bit indices is less than that of LZW encrypted with AES and Salsa20, with squeeze being generally between 10% and 60% slower for compression with a typical slowdown of around 20%. For decompression, squeeze is slower by 30% to 67%. For 16-bit

[26] This is the simplest dictionary management scheme. Note that even when the dictionary is full, we still randomly swap each index that is output by the encoder.

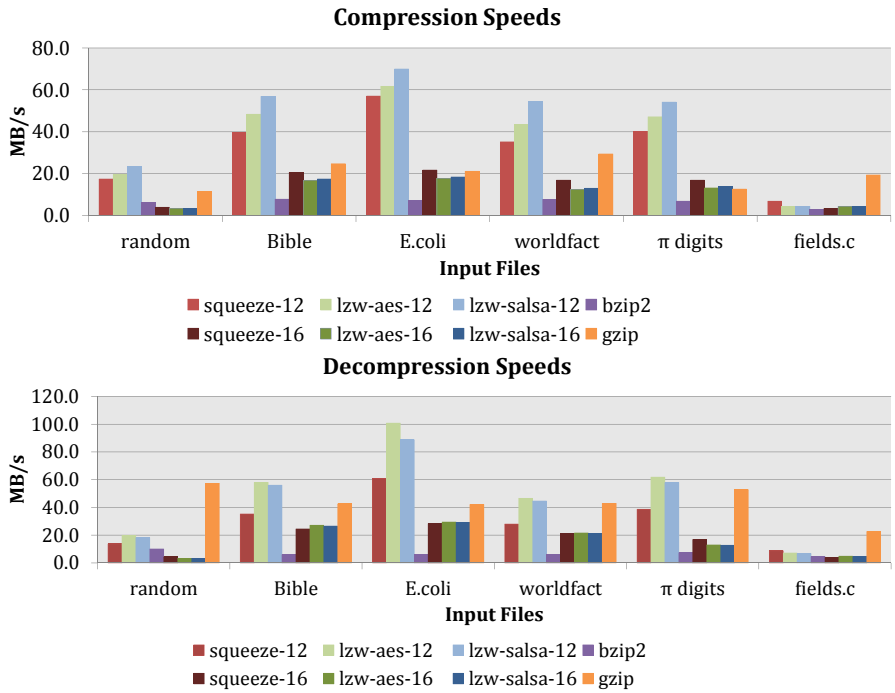


Figure 2.5: Comparison of the speed of compression (top) and decompression (bottom) in squeeze to standard compression algorithms and LZW with AES and Salsa20 encryption. Numbers are in megabytes per second.

indices, squeeze is, in general, slightly faster than LZW with AES and Salsa20 for compression. But, this is due to the fact that squeeze keeps its dictionary half-full while LZW does not. Squeeze then performs fewer dictionary insertions and, on average, has shorter strings in the dictionary, meaning that searching the dictionary takes less time.

Comparing to the standard algorithms, we see that squeeze-12 is faster than both gzip and bzip2 when compressing, and squeeze-16 is faster than bzip2 but generally slower than gzip (though, squeeze-16 is faster on some inputs). For decompression, we note that gzip consistently achieves high speeds while the LZW-based schemes have more variance. LZW with both AES and Salsa20 encryption are generally faster than gzip when decompressing, though squeeze is almost always slower.

Performance Gap. Overall, these graphs demonstrate the practicality of squeeze, in both 12-bit and 16-bit variants, since it achieves high throughput for both compression and decompression. However, it performs slower than the simple compress-then-encrypt schemes based on LZW. The performance gap is due to two factors, one theoretical and the other practical. From a theoretical

perspective, squeeze will always be slower than simply encrypting with a stream cipher since it must use more pseudorandom bits. In particular, suppose we have b-bit indices, then for each output index squeeze must call RandomSwap and use b bits of randomness. If the dictionary is not full and the load is kept to 0.5, then squeeze uses *at least* b bits (with $2b$ bits used on average) in RandomInsert to insert the new entry. That is, when the dictionary is not full squeeze will use at least twice as many pseudorandom bits than simply encrypting with a stream cipher.

From a practical perspective, squeeze must perform more "work" than LZW to accomplish the same task. The function RandomSwap is called on each loop iteration and RandomInsert is called whenever there is still room in the dictionary. These function induce additional reads and writes to memory that are not performed by LZW normally. These extra memory references add up to decrease throughput. To demonstrate this, we ran LZW with both AES and Salsa20 encryption and squeeze-12, with the CIA World Factbook as input and used the `cachegrind` tool of `valgrind` [115] to profile the number of reads and writes performed by each algorithm. The run included both compression and decompression. The results are shown in Table 2.1. It is clear that Squeeze is performing many more read and write operations than LZW both with AES and with Salsa20. The extra data references are due to the additional pseudorandom bits that squeeze must generate and use in addition to the reads/writes performed when updating the dictionary.

Algorithm	Reads	Writes	Total
LZW + AES	107M	14M	121M
LZW + Salsa20	77M	26M	103M
Squeeze	131M	66M	197M

Table 2.1: Number (in millions) of reads and writes performed on CIA World Factbook.

LZW & Salsa20 Buffer Size. The combination of LZW compression with Salsa20 encryption allows for a streaming processing of the data. However, it may be profitable to buffer some of the output indices before encrypting them to achieve better locality and avoid polluting the instruction cache by constantly switching between encryption and compression. We can see in Figure 2.6 that both compression and decompression speeds are only lightly affected by the size of the buffer, with no significant change in speed overall as the buffer size increases. This is due to the fact that Salsa20 can achieve encryption speeds up to several hundred MB/s, while LZW compression is limited to tens of MB/s. Note that Salsa20 encryption also has a small code footprint and small internal state,

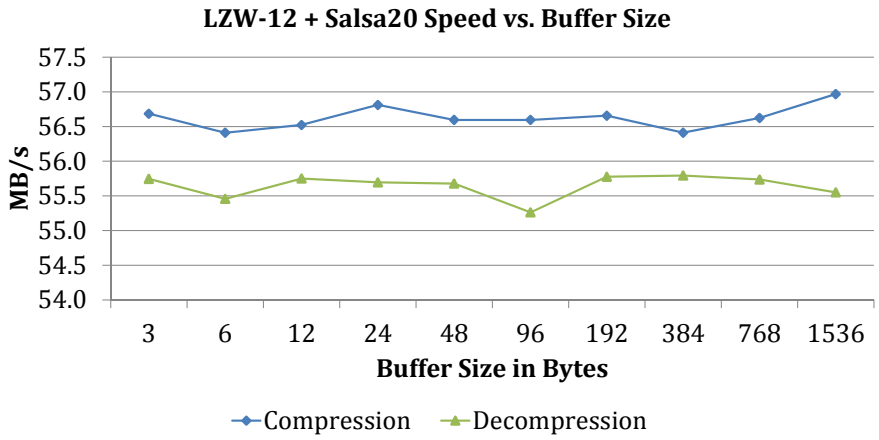


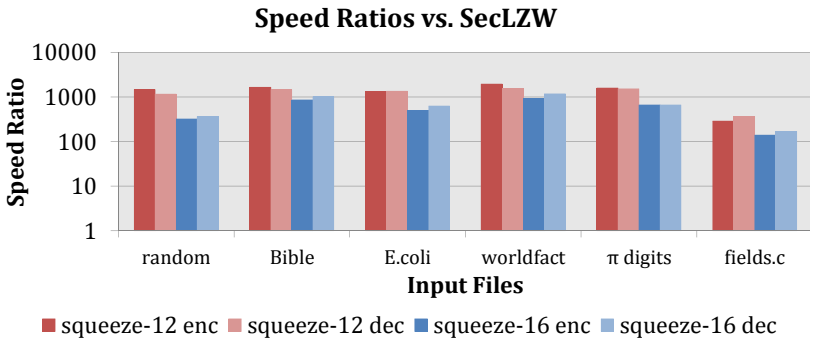
Figure 2.6: Compression and decompression speeds of LZW with 12-bit indices versus the size of the buffer for encryption. Input file is `bible.txt`, the Bible from Project Gutenberg.

so it can easily avoid polluting the caches too much. Thus, the limiting factor in the compression and encryption is the compression itself, making the size of the buffer essentially irrelevant.

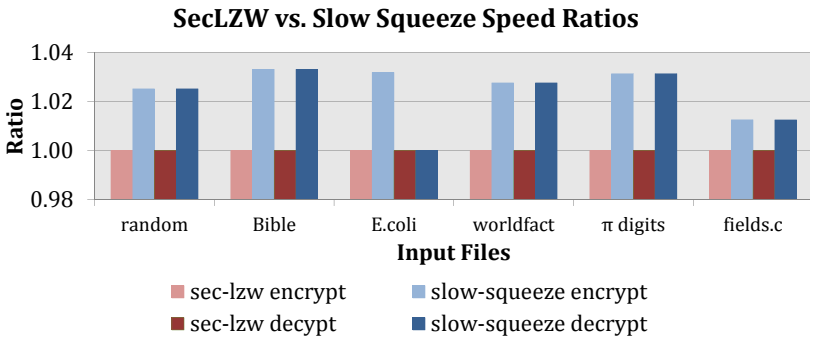
2.6.3 Comparison to Previous Work

There has been much work looking to combine compression and encryption together but most past work has been both inefficient and insecure (see Section 2.8). In [172], Zhou et al. present a secure LZW algorithm—that we denote SecLZW—that is structurally similar to the Slow Squeeze; and so, we compare the it to both Fast and Slow Squeeze here. In their scheme, new entries are randomly inserted into the dictionary (though, they assume there are no collisions during insertion) and then a partial permutation is applied to the dictionary by regarding the dictionary as a square grid and cyclically shifting the rows and columns by random amounts. See Section 2.8.1 for more details.

We compare both Fast and Slow Squeeze to their scheme and show the efficiency gains of Squeeze. We do not compare Squeeze to the LZ78-based scheme in [171] since the level of security that scheme is quite low (see Section 2.8.1 for details) and the comparison would not be meaningful. Figure 2.7a shows the ratio of Fast Squeeze's throughput to the throughput of SecLZW. As can be seen, Fast Squeeze is 2–3 *orders of magnitude* more efficient than SecLZW for both 12 and 16-bit indices. In Figure 2.7b, we can see that Slow Squeeze is also more efficient than SecLZW, though the gains are much more modest, amounting to 1–3%. Note, also, that Slow Squeeze provides much stronger, *provable* guarantees of security than SecLZW in addition to being more efficient.



(a) Ratio of the throughput of Fast Squeeze (12 & 16-bit indices) vs. SecLZW (12-bit indices). Higher is better.



(b) Ratio of the throughput of Slow Squeeze vs. SecLZW, both with 12-bit indices. Higher is better.

Figure 2.7: Comparison of Fast and Slow Squeeze to SecLZW.

2.6.4 LRU Eviction

Here we present our results when implementing a least-recently-used (LRU) eviction policy for the dictionary. In the case of LZW-based compression, the term "used" does *not* mean solely those entries whose corresponding indices have been output. In particular, each time a prefix of the remaining input is matched with an entry in the dictionary we must mark the entry as used *even if its index is not output*. To see this, note that if we remove an entry e from the dictionary, we must also remove *every entry* that has e as a prefix since those entries can no longer be matched. Marking an entry used even when it is not the final match ensures that we do not prune large portions of the dictionary—especially since short strings (e.g., 1 or 2 characters) will be matched often early during encryption and decryption but less so later in the process and risk being evicted.[27]

[27] Removing a single character string can cause the compression to fail (i.e., exit with an error) if that character ever appears later in the input.

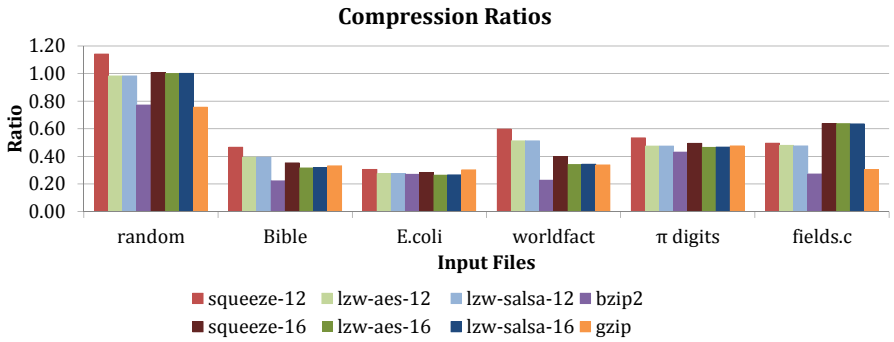


Figure 2.8: Compression ratios when using LRU eviction.

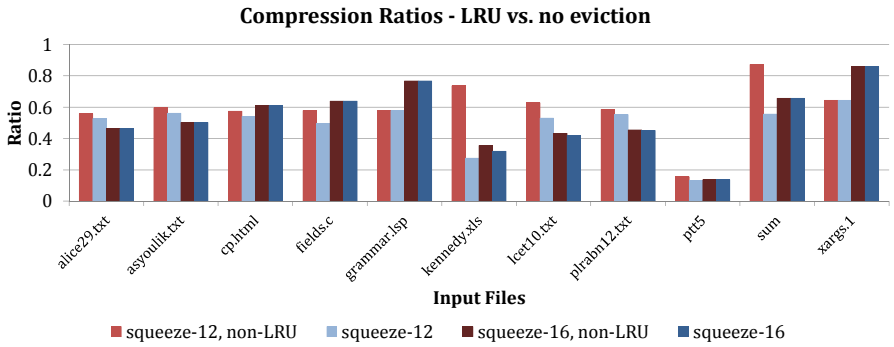


Figure 2.9: Comparison of compression ratios for squeeze using LRU eviction and no eviction.

Compression Ratio. In Figure 2.8, we show the compression ratios for the schemes using the same files as in Figure 2.4 with similar results. We note that LRU eviction does not result in superior compression in all circumstances: for instance, the file of random digits (left-most column), when encrypted using squeeze with 12-bit indices, had *worse* compression when using LRU eviction. (The decrease in compression from eviction is most apparent in LZW using 12-bit indices.) However, for the remaining files, LRU eviction did at least as well and often better (e.g., for the CIA World Factbook).

Figure 2.9 is a graph comparing the compression ratios of squeeze on a subset of the Canterbury corpus, with and without LRU eviction.[28] One notable aspect in the graph is that the use of 12-bit indices generally *improves* the compression of the data compared to using 16-bit indices. This is expected since the schemes can continually adapt to the input. That is, rapid adaption to the input characteristics can give better compression than using a large dictionary that can match longer strings. Note, however, that this does not hold for all of the inputs. Specifically, for the input files `plrabn12.txt` and `lcet10.txt`, the use of 16-bit indices provides better compression.

[28] Specifically, the subset is the complete previous version of the corpus; the new version includes more files.

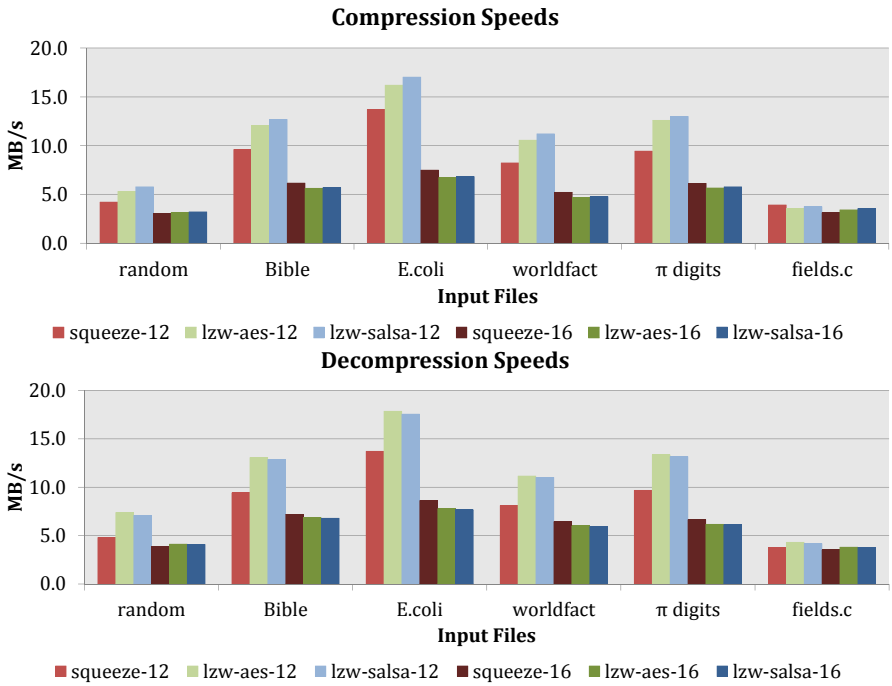


Figure 2.10: Compression and decompression speeds while using LRU eviction.

Speed. In Figure 2.10, we have the compression speeds for squeeze and LZW when given the same files at input as in Figure 2.5 using LRU eviction (while Figure 2.5 has no eviction). Contrasting the speeds in these two figures, we can see that the throughput for LRU is much lower. When implementing the LRU policy, we used a linked list of the entries ordered by recency of access, with the most recent first. Finding the least recently used entry was then simply of a matter of removing the tail of the list. The list was implemented with an intrusive doubly-linked list allowing for constant time insertion and removal. However, even though these operations were constant time, they must be performed for *each character of the input*, resulting in an accumulated overhead that becomes significant. It is worth noting that, as before, the use of 12-bit indices results in a large speed-up.

2.6.5 Squeeze on ARM

We also benchmarked squeeze on a Raspberry Pi running an ARM11 processor at 700MHz with 512MB of RAM, with Raspbian Linux (based on Debian wheezy) as the operating system. Compression ratios are omitted since they are *data* dependent and not architecture dependent. In Figure 2.11 we see the performance of Squeeze versus LZW with AES and Salsa20 on the ARM11 processor. The results are similar to those in Figure 2.5, but the numbers are an order of magnitude smaller.

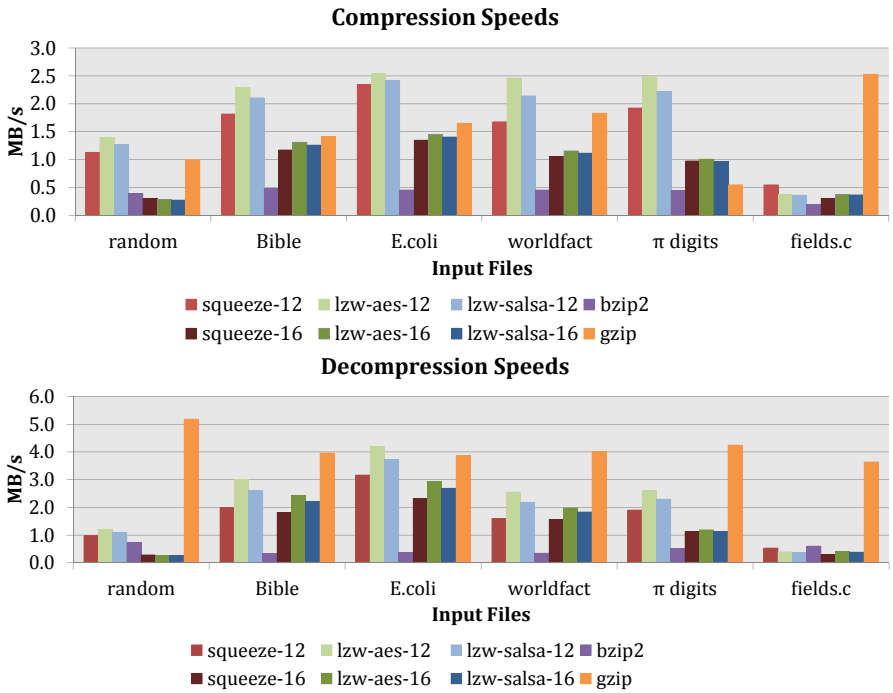


Figure 2.11: Compression and decompression speeds for an ARM11 processor with no eviction.

Worth noting, though, is that LZW combined with AES encryption is the best performing for both compression and decompression even beating LZW combined with Salsa20. This is likely due to code-tuning in the OpenSSL implementation of AES for ARM while the Salsa20 code was simply compiled from untuned code using `gcc`.

Also, as before, squeeze along with plain LZW compare favorably to both bzip2 and gzip, with bzip2 outpaced in almost all cases. For most files squeeze performs well and has encryption speeds close to those of LZW. There is little difference in the performance of the schemes on small files when using 16-bit indices. For large files and 16-bit indices, LZW combined with AES has a small performance advantage over squeeze and LZW with Salsa20. On small files, gzip compresses most quickly while the LZW-based schemes perform well on larger files. When decompressing, as before, gzip performs consistently well and, indeed, almost always outperforms the LZW-based schemes. When compared among themselves, the LZW-based schemes perform similarly, though, as with compression, LZW combined with AES achieves the highest speeds.

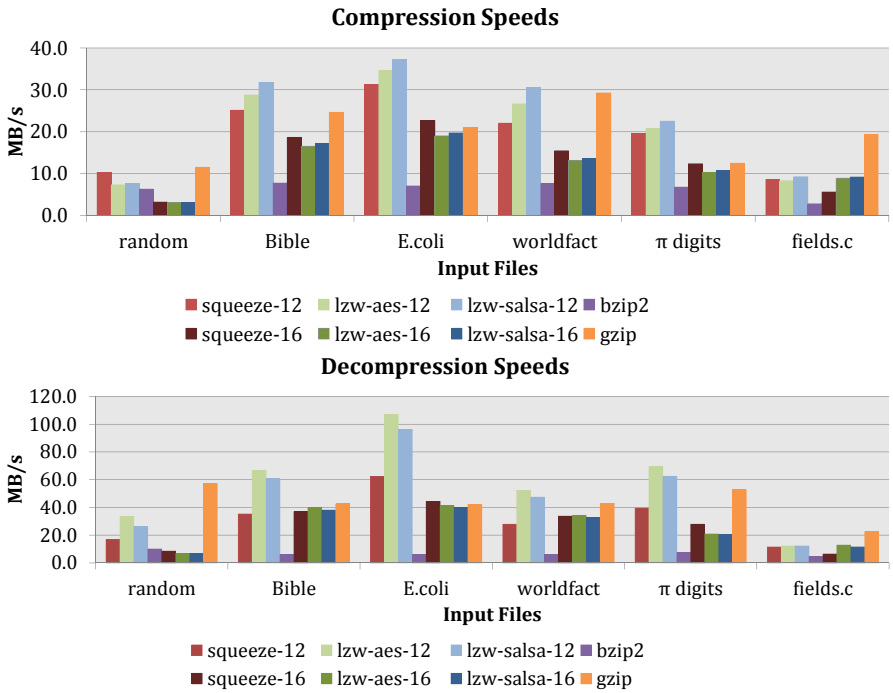


Figure 2.12: Encryption and decryption speeds for squeeze with a compact dictionary.

2.6.6 Memory Efficient LZW

We implemented and benchmarked the memory-efficient versions of squeeze and LZW described in Section 2.4.3. However, to avoid parallel and redundant code for the dictionary, where encryption and decryption would have individualized trie implementations, we implemented a union of the trie nodes described above. Accordingly, some features of the trie nodes were only used when encrypting or decrypting. While this increases the memory requirements of both encryption and decryption, the benchmarks still give us a picture of the performance of the compact dictionary.

Figure 2.12 shows encryption and decryption speeds for squeeze using the compact dictionary on the Core 2 Quad processor. For encrypting large files, the LZW-based schemes match or surpass gzip; on the other hand, for smaller files (such as `fields.c`), gzip performs better than the LZW-based schemes. For decrypting, the LZW-based schemes achieve very high throughput, even exceeding gzip and our more "optimized" (and memory-heavy) implementation above.

Figure 2.13 shows the performance of squeeze with a compact dictionary on the ARM11 processor. The speeds here are similar to those in Figure 2.11, but the speeds shown here are up to 10% *higher* than those in Figure 2.11. This boost in speed is likely due to better cache performance with the

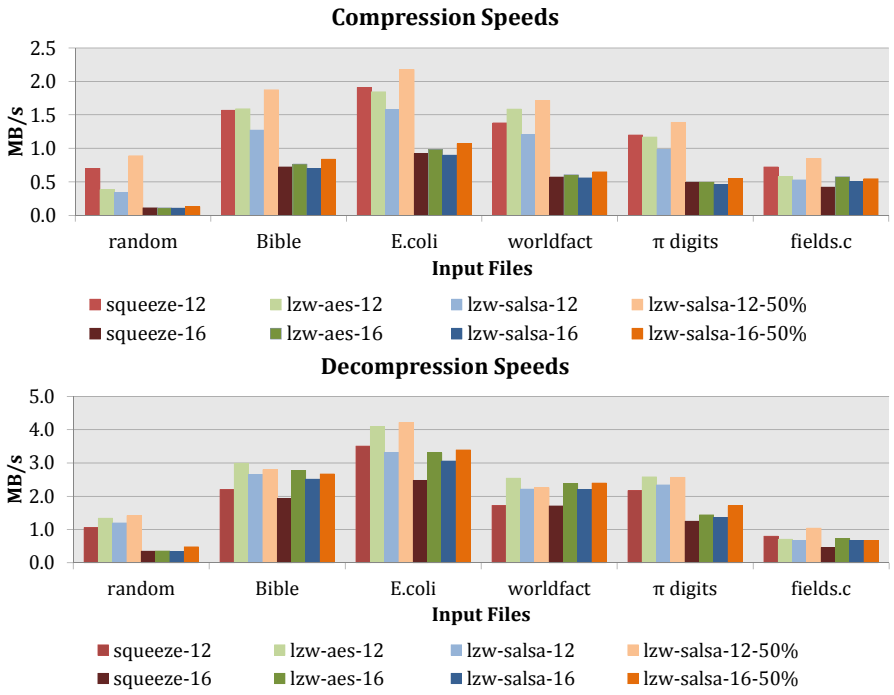


Figure 2.13: Encryption and decryption speeds for squeeze on ARM11 with a compact dictionary.

smaller dictionary. Squeeze fairs well in this comparison having speeds close to and even exceeding those of plain LZW with AES and Salsa20 encryption. However, this can be attributed to the fact that the load in the dictionary is bounded by 1/2. To see this, note that LZW combined with Salsa20 encryption with the dictionary at most half-full achieves the highest compression and decompression speeds across the board.

2.7 Applications and Variants of Squeeze

There are numerous applications and variants of Fast and Slow Squeeze. For instance, if we bound the size of the dictionary and stop encrypting when the dictionary's size reaches the maximum, then we have a cipher that produces outputs up to a fixed size. With appropriate padding, the cipher can produce ciphertexts of constant size. This is similar in operation to a block cipher. However, our cipher would have a variable-sized input block and a fixed-sized output block, while block ciphers have fixed-sized input and output blocks. The size of the input block would depend on the compressibility of the input. Such a configuration would work well with block-oriented storage devices (e.g., a hard drive) since the fixed output size can allow for more efficient writes.

This scheme, however, would not support block cipher modes of operation (e.g., CBC) since they all rely on a fixed block size. But, it is possible to create a chaining mode similar to CBC. In particular, suppose we have some input that will be broken up into several compression operations. The nonce for the compression of the first block is generated at random. The nonce for each subsequent block could be the hash of the previous ciphertext block, using a collision-resistant hash function.

An alternative way to use our compressing cipher would be to integrate several MACs into the ciphertext. Currently with Fast Squeeze, if a ciphertext becomes corrupted it is possible that the receiver will not know about the corruption until it has finished decrypting the entire file. It would be better if the corruption was detected closer to its actual occurrence. A simple way to accomplish this would be to have a maximum input size for the MAC and computing multiple MACs while compressing the data.[29] Note that the data would still be compressed and that the dictionary would *not* be reset whenever a new MAC was started. This is in contrast to the previous variant where the dictionary was reset whenever a new MAC was started (i.e., whenever a fixed-size block was finished).

In Fast Squeeze, we use a single key for computing the MAC for each encrypted message. An alternative would be to have a unique MAC key for every message where each ephemeral key is derived in a process similar to that of the seed derivation process for the PRG. This construction allows us to have a wider selection of possible MAC schemes to use as the MAC only needs to be unforgeable for a *single* message instead of many messages. Furthermore, our session key derivation process in both Slow and Fast Squeeze, where we encrypt a nonce using a PRP, can be replaced with any secure key derivation function (such as those in [27]). Using a password-based key derivation function, such as [74], would allow users to easily encrypt and decrypt files using a password.[30]

2.8 Previous Work

Compression has featured as an important part of many systems, often being used as a pre-processing step before encryption. And consequently, the security implications of compression have been investigated many times. Looking at the threat posed by compression, it has been found that the compressibility of a text can reveal information about the text itself. For example, in [80] Kelsey points out that, contrary to folk-wisdom, adding compression as a pre-processing step for encryption can *weaken* security instead of enhance it. In particular, knowledge about the input and output

[29] The MACs would be output immediately after the block that they authenticate.
[30] The nonce would be combined with the password to produce the session key.

lengths leaks some information about the plaintext. For example, a text document full of English text will compress quite differently from, say, an executable. More powerfully, when using a compression algorithm that adapts itself to the input, an adversary may only know a prefix of the input (e.g., packet headers) and be able to use the compressibility of the remaining text to learn partial information about the unknown portion. This work was later utilized by Rizzo and Duong in an attack on TLS that can steal secret authentication cookies used by websites that employ both compression and encryption [139]. The attack in [139] was later extended by Prado et al. in [131] to exploit HTTP compression in HTTPS sessions.

2.8.1 Cryptography and Plain Compression

Several popular compression schemes include encryption as a post-processing step, including PKZip, WinZIP, WinRAR, and SecureZIP. PKZip used a custom stream cipher that was broken in [18] (the attack was later improved in [153]); WinZIP uses a combination of AES-128/256 and HMAC-SHA1, which was shown to have several weaknesses in [85] detailed below; WinRAR uses AES-128; and SecureZIP can use either AES-256 or triple DES.

There have been several attacks on these compression tools: in addition to PKZip, WinZIP compression and encryption has been found vulnerable. Tadayoshi Kohno in [85] outlines several attacks on the encryption and authentication provided in WinZIP. WinZIP uses AES encryption in counter mode combined with the HMAC-SHA1 message authentication code.[31] The weaknesses are manifold. First, the metadata is not authenticated, so there is a rollback attack that can force WinZIP to use an older (less secure) encryption method for decrypting the data. There is a chosen-ciphertext attack where given an encrypted and compressed file F create by a user U, an attacker $\mathcal{A}$ can XOR it with a pseudorandom pad to produce F'. If $\mathcal{A}$ can convince U to decrypt and decompress it and send the result (e.g., "The file you sent was garbage! Here it is."), then $\mathcal{A}$ can simply re-compress it (since compression is deterministic), remove the random pad and decompress again to recover the original file. There are numerous other attacks described in [85], including key-reuse and dictionary attacks being much more feasible than expected.

[31] This combination is a provably secure authenticated encryption scheme.

Huffman Codes

Huffman encoding is an entropy-encoding scheme that assigns bit-strings to input character strings based on the frequency of those character strings, with the most probable strings being assigned the shortest bit-strings. The code is a prefix code and also optimal. In [47], Gillman et al. analyze the difficulty of decoding a Huffman encoded file without having the prefix tree used to encode the file. They focus on unambiguously decoding the file, which is shown to not be possible in certain situations. However, they do not consider partial recovery of the input file and their own analysis suggests that many files can be partly (but not necessarily fully) decoded.

Wu and Kuo in [170] seek to combine compression primitives with some randomness to produce secure compression without sacrificing speed or compression ratio. They achieve this by applying Huffman encoding with multiple encoding trees where an optimal tree is generated and then mutated in different ways to produce different trees. For example, swapping nodes at the same depth in the tree will not affect coding efficiency. They then use a secret random ordering of the trees to encode sequential input symbols. They apply a similar construction to arithmetic encoders, where they have multiple statistical models for the input and iterate through them repeatedly in a random order. Their security analysis focuses on key recovery attacks and does not analyze possible information leakage from the ciphertext. This is unfortunate since one possible application they list is in digital rights management: where attackers seek plaintext extraction and not necessarily key recovery.

Burrows–Wheeler Transform

The Burrows–Wheeler transform, first described in [22], is an invertible partial-sorting algorithm that permutes the input characters so that identical characters are (roughly) grouped together. Burrows and Wheeler then utilize move-to-front encoding[32] followed by Huffman encoding to compress the data. This algorithm is implemented in the popular bzip2 utility [144]. Külekci in [94] proposes a modification of the Burrows–Wheeler transform to provide privacy. Specifically, he utilizes a secret, randomly chosen ordering of the input alphabet for the partial-sorting step and then uses a separate secret ordering of the alphabet for the move-to-front encoding step. This algorithm was shown to be weak by Stanek in [152], succumbing to both chosen- and known-plaintext attacks.

[32] Move-to-front encoding works by outputting the index of a character in an alphabet, and then moving that character to the front of the alphabet. In this way, if a subset of characters appear frequently, then they will be encoded as small numbers. For example, the string `baba` would be encoded as 2222.

LZ-based Codes

Xie and Kuo in [171] propose a secure compression algorithm based on the LZW algorithm. They start with a randomly permuted initial dictionary and each new entry is inserted into the dictionary in a random position. However, the scheme has several drawbacks. First, they base their algorithm on the LZ78 compression algorithm, which LZW is derived from. The difference between LZ78 and LZW lies in what happens when a match in the dictionary is not found. In LZW, the index of the last matched entry is output; in LZ78, the index of the last matched entry is output along with *the character that failed to match*. That is, the output of LZ78 is an alternating sequence of dictionary indices and individual input characters. Xie and Kuo do not take into account these single characters and so their ciphertexts always leak part of the plaintext. If instead they used LZW, which would be a small modification of their scheme, then this plaintext leakage is mitigated.

However, the scheme still suffers from a critical flaw. Specifically, the scheme does not re-randomize a dictionary entry after it has been used. As long as an entry is never a "target" of a collision during insertion, the corresponding string *always* maps to the same output index.[33] This weakness turns the scheme into, essentially, a substitution cipher. Though, Xie and Kuo add an additional complication to the encryption process by having E different dictionaries (increasing memory use by a factor of E). When inserting a new entry e, they randomly partition the set of dictionaries into two disjoint sets, choose two random numbers r_1 and r_2, and then insert e into position r_1 in one set and in position r_2 in the other set. While this complicates the algorithm a bit, it does not change the fundamental substitutionary character of the cipher itself.

As an example, consider the two message aa and ab: when encrypting the aa, the ciphertext, with high probability, will consist of two *identical* indices since they do not update the dictionary after using an entry. In particular, when inserting aa, with probability $(1/E)(1 - 1/|D|) + (E - 1)(1/|D|)$ (where $|D|$ is the size of the dictionary) aa will *not* collide with a. This means that when the second a is matched the same index will be output with probability at least $(1 - 1/|D|)(1/E)$. On the other hand, the string ab will encrypt to two *different* indices with probability $1/E + (1 - 1/|D|)^{E-1}$. In particular, when encoding ab, for a and b to have the same index, b would need to be placed in a's entry after encoding a, which is not possible. Namely, when ab is inserted, if it lands in an occupied entry, that entry is evicted and placed at the *end* of the dictionary: ab is then inserted into the

[33] When inserting a new entry, the scheme chooses a random place in the dictionary and evicts whatever entry may be there. The evicted entry is then appended to the end of the dictionary. This is essentially a single step of the Fisher-Yates algorithm [42].

now-empty position. So there is no way for b to occupy a's entry since it can only stay in place or be moved to the end. Thus, the scheme cannot provide even ER-CPA security since the messages ab and aa are distinguishable with non-negligible probability.

Another drawback is that they handle collisions in the insertion procedure by evicting the occupying entry and moving it to the end of the dictionary (extending the dictionary by one entry). When the dictionary is full—which can happen since they use finite-length indices—this is equivalent to evicting the old entry from the dictionary. This can cause compression to fail as the eviction process may remove one of the initial single character strings and then the compressor will no longer be able to encode that character if it is subsequently encountered. Finally, the security analysis in [171] is a bit narrow and only considers key recovery attacks and rather than information leakage and/or indistinguishability of ciphertexts.

In [172], Zhou et al. propose a secure LZW algorithm using a randomized dictionary similar to Fast and Slow Squeeze. They start by initializing the dictionary randomly, i.e., putting each single character string into a random entry. When inserting a new entry, they use a keyed hash function to choose a new, random location for the new entry. After encoding a string, they apply a random partial permutation to the dictionary. There are a few short comings of this algorithm. Firstly, it is never stated how collisions are handled when an insertion is performed; it seems to be assumed that there are no collisions. But, due to the birthday paradox, if they have a dictionary of size 2^b, then a collision will happen with probability greater than 50% after $2^{b/2}$ insertions.

Secondly, their security analysis focuses on adversaries that are seeking to recover the key and do not properly consider known- or chosen-plaintext attacks (while they mention chosen-plaintext attacks, it is in the context of key recovery and not privacy). As a result, their scheme is vulnerable to chosen-plaintext attacks as detailed by Li et al. in [96].

Thirdly, the permutation step they apply after each encoding step is quite expensive. The permutation process is equivalent to adding the numbers $s_1 2^{b/2} + r_1$, $s_1 2^{b/2} + r_2$, $s_2 2^{b/2} + r_1$, and $s_2 2^{b/2} + r_2$ to different quarters of the dictionary D. This requires $O(|D|)$ operations to be performed at *each* encoding step, increasing the work of encoding and decoding by a factor of $|D|$, drastically reducing the scheme's efficiency. See Section 2.6 for an experimental comparison of the work of Zhou et al. with Fast and Slow Squeeze.

Li et al. in [96] give possible remedies to the security problems of [172] including enhancing the dependence of the ciphertext on the input plaintext by, e.g., hashing the to-be-inserted string with

the index of the previously inserted string to produce the new index. They also suggest including an initialization vector to prevent the chosen-ciphertext attack on single-character strings. But, these enhancements still suffer from the same profound inefficiency of the original scheme.

Arithmetic Coding

Arithmetic coding compresses data by dividing the interval $[0, 1)$ into disjoint segments whose lengths correspond to the probability of a particular character occurring in the input stream. The algorithm reads a character, sets the current upper and lower bound to the end points of the associated interval and then recurses on that interval. At the end of the input stream, we have an interval that uniquely determines the sequence of input symbols, and the final output is the shortest element in that interval. This encoding scheme forms a prefix code and produces outputs that are close to optimal in size. The probability distribution over the input (called the *model*) can be fixed or dynamic (e.g., adaptive to the input).

There have been several efforts to combine arithmetic coding with encryption. One example is [81] where Kim et al. split each interval into smaller segments and then permute the segments all together. In this way, each symbol has an associated set of subintervals rather than a single, continuous interval. When outputting the final string, for each set of subintervals corresponding to an input symbol, the encoder selects one of the subintervals and uses the shortest string from it. Their scheme also includes applying a permutation to both the input data and the final output strings. Note that this precludes their cipher being used on streaming input without buffering the input to be encoded. Also, the permutation itself is not a random permutation but consists of cyclic shifts of even and odd rows and columns where even/odd columns/rows rotated the same amount. The work of Zhou et al. in [173] provides a chosen-ciphertext attack on this scheme that can remove the final permutation in time linear in the number of chosen ciphertexts and then and then break output of the encoder.

In [29], Cleary et al. present known- and chosen-plaintext attacks against a simple arithmetic encoder with a fixed (but unknown) probability distribution on the input symbols. The unknown distribution acts as they key for the system. They show that they can recover the distribution with a chosen plaintext of $w + 2$ characters, where the distribution is calculated with w bits of precision.

Witten and Cleary in [167] present two similar schemes for secure, adaptive arithmetic coding where the encryption key is the model used for the encoding. In the first scheme, the key is the initial (random) model of encoder. In the second scheme, the encoder starts with a fixed model but

ingests a random string (i.e., as a prefix to the input) to have the model in an unknown state to an observer. In [14], Bergen and Hogan present a chosen-plaintext attack on the first scheme that works by "flooding" the encoder to force it into a known (or more manageable) state and leverage that to synchronize its model with the encoder's and decrypt subsequent messages. In [97], Lim et al. build on [14] and present an adaptive, chosen-plaintext attack on both schemes and can recover the secret key (the attacks are given in full detail by Bergen and Hogan in [13]).

A randomized arithmetic encoder is proposed by Grangetto et al. in [56], where, when processing an input symbol, the partitions of the interval are randomly permuted. They do not address known- or chosen-plaintext attacks on their system. Indeed, it succumbs to a chosen-plaintext attack where they attacker can slowly probe the encoder with incrementally longer plaintexts to learn the order of intervals at each step. Their scheme also does not consider biased input distributions—for example, if we have $P(0) = 0.1$ and $P(1) = 0.9$ at each step, then the encryption of any message will necessarily leak information about the plaintext. In particular, it is impossible for any input starting with 0 to produce a ciphertext that is in the interval $[0.1, 0.8)$.

Since their proposed application scenario is in selective encryption of multimedia files, partial information leakage is not considered that deeply. Indeed, an attacker can simply apply the standard arithmetic decoder to a ciphertext and recover partial information about the input. This is shown in Figure 7 in [56] where they decode an encrypted Lena picture and the result clearly shows outlines of elements in the image (i.e., it is readily apparent that the image is of a woman wearing a wide-brimmed hat). There is, however, a significant degradation in the quality of the image.

In [168], Wong et al. present a scheme based on the observation by Bogdan et al. in [105] that the reverse application of the skew tent map is equivalent to arithmetic coding. In particular, they combine the skew tent map over the unit interval with a PRG that is also based on the skew tent map. In the encoding phase, they use the pseudorandom bit stream to circularly shift the subintervals and (possibly) reflect each subinterval around its center-line. They XOR the output of the coding phase with some more pseudorandom bits produced by an integer tent map. They note that the finite precision integer tent map does not possess sufficient randomness to use the output directly, so they only output a few of the least-significant bits. They evaluate the strength of the PRG by running it against the NIST Statistical Test Suite [140] and passing, but perform no further analysis. The security analysis of the overall cipher is a bit sparse and contains several heuristic (rather than formal) arguments, leaving the overall security of the scheme rather uncertain.

Another chaos-based secure arithmetic coder is presented in [169] by Wong and Yuen, where they use a logistic map with key-dependent parameters to perform the compression and encryption. The interval $[0, 1)$ is divided into subintervals which are partitioned into disjoint random subsets. Each subset corresponds to an input symbol with sizes proportional to the likelihood of the symbol. The key-dependent logistic map is applied to a secret initial value iteratively. Each input symbol is mapped to the number of iterations needed by the logistic map to "land" in one of the subintervals in that symbol's set of subintervals. Wong and Yuen are able to achieve mild compression with this scheme (e.g., 25% compression for `book2` from the Calgary corpus), but the security provided is minimal. There does not appear to be any way to update the mapping of a symbol to its number of iterations, so the cipher reduces (almost) to a substitution cipher (some symbols are encrypted differently, and this only moderately increases security—see the original paper [169] for more details).

2.8.2 Authenticated Encryption

The combination of encryption and message authentication has provided confidentiality and integrity in a wide variety of deployments (e.g., SSH and TLS). This work utilizes the Encrypt-then-MAC (EtM) paradigm described by Bellare and Namprempre in [10] in the construction of the Fast Squeeze cipher. EtM is known to be IND-CCA-2 secure, which—given the observation in Section 2.2.3 that results independent of ciphertext length also apply in our entropy-restricted context—provides an alternative proof that our constructions are also ER-CCA-2 secure. Authenticated encryption itself is a valuable tool, and there has been much research to analyze general constructions (e.g., [10]) as well as specific constructions—for instance, there are several block cipher modes developed to provide authentication of the message, see [11, 86, 110].

2.8.3 Length-hiding Encryption

In Section 2.2 we define ER-CPA/CCA-1/CCA-2 security for our scheme and require that the two challenge message m_0 and m_1 must come from the same class of message $\mathcal{C}_{l_1,l_2} \subseteq \mathcal{M}$, where all $m \in \mathcal{C}$ compress to sizes within the range $[l_1, l_2]$. This is done so that the length of the final ciphertext does not leak information about which message was encrypted. Of course, if $l_1 < l_2$, then we must add random-padding to ciphertexts of the short messages in $\mathcal{C}$ so that their lengths will be the same as that of the longest compressed-encrypted messages in $\mathcal{C}$. That is, the ciphertexts must be

self-delimiting in a way that is not visible to the adversary, e.g., include some EOF marker in the plaintext and then append random bits.

There has been past work looking at padding schemes for hiding plaintext length. For instance, Paterson et al. in [123] define *length hiding authenticated encryption* (LHAE) which provides secure authenticated encryption while using variable length padding appended to the plaintext. Their work focuses on the MAC-Encode-Encrypt (MEE) paradigm (i.e., MAC, pad, encrypt), which is used in TLSv1.2. They prove that if the size of the MAC tag is larger than the block size of the cipher, then MEE is secure LHAE. This work was later used by Jager et al. in [69] to prove the security of the TLS handshake protocol.

The encryption function for an LHAE instance takes a length parameter ℓ as input which is the target output length for the ciphertext. The two input messages are allowed to be of arbitrary length as long as their encryptions succeed (i.e., do not produce an error), placing an implicit restriction that ℓ is greater than or equal to the lengths of the input messages. Note that this ℓ parameter is *independent* of the message content encrypted; for ER-CPA security, the length of the output ciphertext is *dependent* on the contents of message (in particular, dependent on its compressibility).

In [162], Tezcan and Vaudenay examine the security provided by adding padding to the end of a message. Specifically, they consider an adversary that seeks to distinguish the combined padding and encryption of two messages which are allowed to be of different lengths (with a bounded difference). They prove that *in*security decreases *linearly* with padding length and that selecting the padding length uniformly at random is nearly optimal. This is in contrast to our work, where if two messages m_0 and m_1 are in a class $\mathcal{C}_{l_1,l_2}$, where $l_1 \neq l_2$, then we apply padding to the ciphertext (and append a special EOF marker to each message) to ensure that the ciphertexts have exactly the same length.

2.9 Future work

There are several directions for future work with secure compression. First and foremost, while it is natural for data (e.g., a file) to be read in a sequential fashion, this is by no means the only (or even primary) way of accessing data. Often, users only need a portion of a file rather than all of it and would like to seek within a file (sometimes randomly), skipping the unneeded segments. Developing a *random access* variant of the squeeze ciphers would be synergistic with many file access patterns. Random access is possible with block ciphers (e.g., counter mode) and compression functions

(see [90]), so intuitively there does not appear to be anything inherently precluding a compressing cipher from also having this property.

Second, as noted by Kelsey in [80], the *compressibility* of a message leaks information about the contents of that message and it is desirable to address this leak. While our model restricts itself to comparing plaintexts that have the same compressibility, in real-world applications users do not always have that luxury. Indeed, our solution for this required a special EOF symbol in the alphabet, which is not always possible when, for example, compressing binary data where adding a special symbol would increase the size of *every* symbol by a bit and undermine the compression. One possible, alternative, way to relax this restriction is to try and add some "noise" to the compression process, either via: (1) insertion of random data into the input stream in pseudorandom positions, determined by the key (similar to the work of Witten and Cleary in [167]); or (2) prepending/appending a variable-length random pad to the message (as in [123]) or to the ciphertext as described in Section 2.2.3

Finally, another direction of work is to explore alternative compression frameworks and techniques. Here, we provided a construction based on a self-constructing dictionary, alternatives include prediction by partial matching [30], grammar-based codes [116], and integrating a variant of the Burrows–Wheeler transform. And of course, additional optimizations in the compression and efficiency of the squeeze ciphers would be worthwhile. Part of this effort would be to devise a parallelizable variant of the algorithm to take advantage of multi-core systems.

2.10 Conclusion

In this chapter we have presented the first theoretical framework for analyzing combined compression and encryption schemes. The definition of entropy-restricted semantic security (ER-CPA security) and its refinements (ER-CCA-1 and ER-CCA-2) are based closely on the standard definitions of security for ciphers; and, indeed, they are shown to be a particular generalizations of them. We also presented two keyed compression algorithms—both simple, with one being particularly efficient—and we proved that they both achieve the strongest form of entropy-restricted security. Additionally, we outlined several alternative constructions and variants of the schemes which have their own interesting properties, and then finally, we presented several future directions for the work.

CHAPTER **Three**

Authenticated Error Correcting Codes

3.1 Introduction

Over the last few decades, the proliferation of computing systems to near ubiquity has had manifold impacts on our world. Among those is a rapid growth in the production of electronic data as well as an ever growing appetite for that same data, i.e., the aptly named Big Data. Of course, this data must be stored somewhere. Moreover, much of this data is of a sensitive and personal nature—such as electronic healthcare records—and the possible corruption and loss of this data can have serious consequences for both individuals and organizations (see, for instance, [151]). Hence, strong guarantees of system reliability and data integrity are necessary in modern computing systems.

Reliability and integrity can be achieved in a number of ways. The simplest method is to replicate the data in multiple locations so that if any disk drive or file server crashes/fails/catches on fire, another copy of the data is readily available. However, this technique is quite wasteful since, if data is replicated r times, it will require an r-fold increase in the amount of storage, and hence, the effective capacity of the storage system is reduced by a factor of r.

An alternative technique is to add some specially constructed redundancy to the data so that if a portion is lost, it can be reconstructed. For instance, in RAID5 systems,[1] the data is broken up into fixed-sized blocks (typically 4KB) and then distributed among several disk drives. One of the disks is a dedicated *parity disk* where each block is the XOR of the corresponding data blocks from the other

[1] RAID standard for "redundant array of inexpensive disks" and is a standard technique of combining several disks together to achieve higher system throughput and reliability. There are different configurations numbered RAID0 through RAID6.

disks. For instance, the first block of the parity disk will be the XOR of the first block from each of the other disks. This configuration allows the loss of a single disk without losing any data as the missing disk can be reconstructed by simply taking the exclusive-or of the other disks' contents. If there are 5 disks in a RAID5 array, then 4 store data and one stores the parity information, giving an overhead of just 25%, much less than simple replication of data.

A more powerful and flexible technique to achieve reliability is to use an *error correcting code* (ECC), a mathematical transformation of data such that the encoded output can withstand some amount of mangling/corruption and still allow for the recovery of the original data. Codes that are resilient to only data *loss* are called *erasure codes* (such as RAID5). Some codes allow for including the input message directly in the encoded output for greater encoding and decoding efficiency: such code are called *systematic*. Additionally, if we allow the decoder to output at list of values, instead of a single value, then we call this a *list-decodable code*. Often, list-decodable codes can correct *more* errors than their unique-decoding counterparts. In comparison with the RAID5 example above, we note that a RAID can only detect single block corruption but cannot recover from data corruption.[2] Additionally, errors in multiple blocks may cancel out when combined with XOR and remain undetected.

Reed-Solomon codes [138] are error correcting codes that, due to their simple description and practicality, have found many applications—including error correction of transmissions in space exploration (e.g., the *Voyager* spacecraft [166]), computer storage systems such as HDFS-RAID [135], and CDs and DVDs (see [166] for additional applications). Reed-Solomon codes operate by breaking up the data into k, equal sized pieces (called *message symbols*) and regarding these pieces as coefficients of a degree $k-1$ polynomial $p(x)$ over some finite field $\mathbb{F}$ (where each symbol is an element of $\mathbb{F}$). The encoder then evaluates $p(x)$ at $n \geq k$ distinct points in $\mathbb{F}$, generating n *code symbols*; these code symbols form the encoding of the data. Using polynomial interpolation, the original data can be recovered from any k out of n uncorrupted code symbols, tolerating up to $n-k$ erased symbols. If some of the data is corrupted, then the decoder needs $k+(n-k)/2$ uncorrupted code symbols to decode correctly; that is, the code can tolerate up to $(n-k)/2$ corrupted code symbols (where the corruption of a symbol can be arbitrary).

The gap between the erasure and error correction capacities of Reed-Solomon codes comes from the fact that when a corrupted codeword arrives, the decoder does not know which symbols are corrupted.

[2] The mismatch between the parity blocks and the XOR of the data blocks will signal that data is corrupted, but it is impossible to tell which block is the bad one.

If the decoder has this information, then it can simply discard the bad symbols, regarding them as erased, and only work with the good symbols. This closes the erasure-error gap and the decoder can then correct $n - k$ errors. Locating the corrupted symbols can be achieved by simply computing a checksum over each symbol and appending it to the symbol. This is a well-known technique and has been used for at least two decades.[3] Popular checksum algorithms such as CRC, Adler's checksum, and Fletcher's checksum are best suited for *random* corruptions of data and cannot withstand purposeful, *adversarial* corruption of data. Stronger primitives, such as *message authentication codes* (MACs) and *digital signatures*, can provide protection against any (polynomially-bounded) data-corrupting adversary (such as in [88]). We call such augmented codes *authenticated error correcting codes*.

One notable drawback to using checksums is that it is impossible to distinguish between the cases when the data is actually corrupt and when just the checksum itself is corrupt. This ambiguity can lead to situations where perfectly good data is thrown out. One system where this can have a large impact is in large distributed file systems such as the Google File System (GFS) [45] and the Hadoop File System (HDFS) [134]. In these systems, files are divided into chunks that are tens to hundreds of megabytes in size (with typical sizes of 64MB and 128MB) and distributed among several storage servers. Throwing out good data in this context is quite costly since new data would need to be copied from another server and/or reconstructed from an encoding using an error correcting code (see [143] for an analysis of the cost of data repair in HDFS).

Contributions. In this work, we give two efficient constructions that resolve this bad-data/bad-checksum ambiguity. In the first construction, we completely separate errors in the symbols from errors in the checksum by applying an efficient list-decodable code to the checksums. This allows us to correct any errors in the checksums, even up to the theoretical maximum number of errors, and thereby detect up to the maximum number of errors in the code symbols themselves. The second construction takes the opposite approach by tying together corruption of the data and the checksums; in particular, we compute a checksum over the data and encrypt the concatenated result with a non-malleable cipher. In this way, any corruption of the ciphertext will corrupt both the data and the checksum with overwhelming probability. In addition to these constructions, we provide a new security framework and we define a new adversarial model that is more powerful than previously considered. Moreover, we prove the security of our constructions against this strong adversary.

[3] Krawczyk in [87] used this to construct *distributed fingerprints* in the early 1990s.

Organization. The rest of this chapter is as follows. In Section 3.2 we give a summary comparing our constructions to similar, previous efforts. In Section 3.3 we provide basic definitions for the tools that we will be using. In Section 3.4, we describe our new adversarial model and corresponding security definitions. We detail our constructions in Section 3.5 and prove their security in Section 3.6. We then provide an experimental evaluation of our constructions in Section 3.7. Section 3.8 surveys the previous work in this area; and, finally, we conclude with Section 3.9.

3.2 Comparison to Previous Work

The constructions we provide in this work use cryptography to enhance the error correction of various ECCs. There are numerous examples of cryptography and error correcting codes being combined in this way and here we compare our work with representative samples of previous constructions.

Briefly, an error correcting code maps a message consisting of k symbols to a string of $n \geq k$ symbols such that any (uncorrupted) k-subset of the n symbols can be used to reconstruct the original message. The *minimum distance* d of a code is the minimum Hamming distance between all pairs of distinct codewords. If the code is an $[n, k]$ code, then $d \leq n - k + 1$. For example, Reed-Solomon codes reach this bound exactly while a Hadamard code, which has $n = 2^k$ for a given message length k, has distance $2^{k-1} < n - k + 1$. The minimum distance bounds the maximum number of correctable errors for a code by $\lfloor (d-1)/2 \rfloor$.[4]

Sign-all. The most basic scheme is the "sign-all" approach that simply computes a signature or MAC over every code symbol. Any changes to a symbol will be detected with overwhelming probability and the symbol can be discarded. The inclusion of a nonce in the signature prevents replays of old symbols. For an $[n, k]$ code, the signatures allow efficient error detection and so code can correct $n - k$ errors. Examples of works that use this approach include [17] and [25].

Hashing. Another approach is to use the code symbols as the leaves in a Merkle tree. For each symbol, the encoder appends the hashes of the sibling nodes on path from the symbol to the root of the tree. This approach adds an overhead of $\log n$ to each symbol, where n is the number of code symbols. This approach was used in [24] and [76].

[4] Intuitively, this is the maximum radius of a sphere centered at each code word such that none of the spheres overlap.

Table 3.1: A comparison of this work with previous constructions. Assume we are using an $[n, k]$ code to encode the message. The first two rows indicate the number of errors the scheme can correct given a particular adversary where d is the minimum distance of the code ($d-1$ is the theoretical maximum number of correctable errors). Joint and independent corruption indicates whether symbols and authentication takes are corrupted jointly or independently.

	Sign-all	List decoding	Merkle tree	LD	NM
Joint corruption	$n-k$	$d-1$	$n-k$	$d-1$	$n-k$
Independent corruption	$(n-k)/2$	$d-1$	$(n-k)/2$	$d-1$	$n-k$
Uses list decoding	no	yes	no	yes	no
Any message alphabet	no	yes	no	no	no
Any erasure code	yes	no	yes	yes	yes

Another approach that utilizes hashing but does not require a Merkle tree is in [91]. The authors use homomorphic hashing—where security is based on the difficulty of computing discrete logarithms—to authenticate code symbols for reliable and secure content distribution. The message symbol hashes are published by the sender in a reliable and secure way accessible by the receiver. The encoder produces parity symbols by adding together a small, random subset of message symbols and the hashes of the parity symbols are computed by, for each symbol, taking the product of the hashes of message symbols constituting the given parity symbol. The overhead in this scheme is similar to the sign-all approach, but they also give a recursive hashing scheme (where the hashes are, themselves, hashed) to produce more succinct authentication tags at the cost of more computation. (For simplicity, we consider the non-recursive version.)

List Decoding. In [106], Lysyanskaya et al. couple signatures with list decoding to achieve error correction (over a computationally-bounded channel) beyond the unique decoding bound of $\lfloor (d-1)/2 \rfloor$. Specifically, they sign the input message and then apply an efficiently list-decodable code (e.g., a Reed-Solomon code) to the message-signature pair. When decoding, the codeword is list decoded and the signature on each list entry is checked to find the correct decoded message. This technique of using signatures to achieve unique decoding from list decoding over a computationally-bounded channel was independently discovered by Micali et al. in [111]. This latter work also developed the theoretical foundation for a computationally-bounded channel, which we adapt to this work. Note that the error correction is $d-1$, which may be less than the theoretical maximum of $n-k$. This is due to the fact that in these schemes the signature is applied to the message *before* the encoding, so the code itself must absorb all of the errors.

We present two constructions in this work that are secure against a new form of data corruption against authenticated error correcting codes. In particular, an adversary that can corrupt the symbols and authentication tags independently of each other (e.g., corrupt e symbols and corrupt e tags) we call *independently corrupting*. Else we call it *jointly corrupting*. For example, the sign-all approach can only tolerate $(n-k)/2$ independent corruptions since it can cause $n-k$ total symbols to be discarded (i.e., the corrupted symbols and the symbols with corrupted tags).

Our first construction that is secure against an independently-corrupting adversary uses list decoding to completely separate the corruption of the symbols from the corruption of the tags: i.e., so that their respective errors can be corrected individually. We denote this scheme with LD in Tables 3.1 and 3.2. Our second construction applies a pseudorandom permutation (PRP) to a message symbol, computes and appends a MAC to each symbol, and then encrypts the symbol-MAC pair with a non-malleable cipher. This ensures that the symbols and tags are "bound together" such that the corruption of one corrupts the other with overwhelming probability. We prove this in Section 3.6. We denote this scheme with NM in the tables and, for simplicity of presenting the overhead in Table 3.2, count its MACs as signatures.

A comparison of this work with previous work is in Table 3.1. Each row in the table is a particular property of the schemes and each column is a particular scheme. The first row compares the error correction capacity of the scheme against the standard, computationally-bounded, jointly-corrupting adversary, where the adversary must regard a symbol and its authentication tag as a single unit. This is in contrast to a stronger adversarial model (compared in the second row) where adversary is permitted to corrupt symbols and tags independently of each other. Note that both of our constructions are secure against this stronger adversary. The third row indicates which schemes use list decoding, which is generally a costly tool.

The fourth row of the table highlights a notable feature of the schemes in [106] and [111]: both schemes work over alphabets of all sizes (e.g., $\Sigma = \{0,1\}$), but do require a minimum message length of λ, the security parameter. Otherwise if the message length less than λ, the signature scheme cannot provide security with parameter λ. Our LD and NM constructions, for comparison, have a lower bound on the size of the *symbols* used (i.e., $\Sigma = \{0,1\}^{\lambda}$), which places a larger lower-bound on the message size than [106] and [111] do. Note that the sign-all approach requires large alphabets as well.

The last row of the table indicates whether or not the message being sent may be encoded with (almost) any erasure code. The work done in [106] and [111] requires that the message be encoded

Table 3.2: Concrete comparison of the overhead associated with the schemes. Encoding of the message is done with an $[n, k]$ code with rate $R = k/n$. Hashes and signatures are h and s bytes in size (resp.). We have L denote the size of the list for decoding and have t_L denote the time to list decode. t_σ is the time to compute or verify a signature while t_h is the time for a hash operation. t_E is the time for any encoding in addition to encoding the input message. t_{prp} and t_{nm} denote the time to apply a PRP and a non-malleable cipher (and the respective inverses). Let $Q = \log q$ be the size of the encoding alphabet (note, $\lambda < Q$). For simplicity, we omit the cost of nonces and counters. Spaces costs are *per symbol* while computational costs are *per message*.

	Sign-all	List decoding	Merkle tree	LD	NM
Space cost	s	s/k	$h \log n + s/k$	$h/R + s/k$	s
Encoding	nt_σ	t_σ	$t_\sigma + (2n-1)t_h$	$nt_h + t_\sigma + t_E$	$n(t_\sigma + t_{\text{prp}} + t_{\text{nm}})$
Decoding	nt_σ	$t_L + Lt_\sigma$	$t_h n \log n + t_\sigma$	$t_L + Lt_\sigma + nt_h$	$n(t_\sigma + t_{\text{prp}} + t_{\text{nm}})$

with a list-decodable error-correcting code (and, in particular, an *efficiently decodable* code). The other schemes, however, do not have this restriction and the code used on the message itself can be an arbitrary erasure code (as long as the code supports the required alphabet size and has efficient encoding and decoding). Though, note that the LD construction requires an efficiently list-decodable code to be applied to the authentication tags. But, the message itself (which can be much larger than the authentication tags) need not be encoded with a list-decodable code.

One feature omitted from Table 3.1 is the fact that our list decoding solution (LD) does not require us to perform list decoding on the entire message. Rather, we perform list decoding on just the authentication tags. This allows us to perform decoding more efficiently since the decoding is done over a much smaller field. Holding the message length k and codeword length n constant, and letting the symbol alphabet Σ grow, LD has a fixed list decoding cost while the cost of the schemes in [106] and [111] grows with Σ. Note, however, that our LD scheme requires the use of cryptographic hashes, whose cost grows linearly with alphabet size. But, hashing is a very fast operation compared to list decoding, so this growth is not significant.

Table 3.2 compares the computational and space overhead of the different schemes. (A simplified, asymptotic version is given in Table 3.3.) The first row of the table compares the schemes based on the space overhead per symbol. The sign-all and NM schemes both require a signature per symbol. The schemes based on list decoding in [106] and [111] both amortize the overhead of the signature across the message symbols. Our list decoding scheme does the same, in addition to having the, constant, overhead of the hashes—the n hashes are divided into k pieces, giving an overhead of $nh/k = h/R$, where R is the code rate. The Merkle tree-based schemes have, at least, an overhead logarithmic in

Table 3.3: Asymptotic comparison of the overhead associated with the schemes. Encoding of the message is done with an $[n, k]$ code with rate $R = k/n$. Hashes, MACs, and signatures are assumed to have lengths proportional to the security parameter λ. We have L denote the size of the list for decoding and have $t_L(a)$ denote the time to list decode over an alphabet of size a. t_E is the time for any encoding in addition to encoding the input message. Let $Q = \log q$ be the size of the encoding alphabet (note, $\lambda < Q$), and let $\ell = Qn$ be the length of an encoded message. For simplicity, we omit the cost of nonces and counters. Spaces costs are *per symbol* while computational costs are *per message*.

	Sign-all	List decoding	Merkle tree	LD	NM
Space cost	λ	λ/k	$\lambda \log n + \lambda/k$	$\lambda/R + \lambda/k$	λ
Encoding	$O(\ell)$	$O(\ell)$	$O(\ell + n\lambda)$	$O(\ell + n\lambda) + t_E$	$O(\ell)$
Decoding	$O(\ell)$	$t_L(Q) + O(L\ell)$	$O(\ell + \lambda n \log n)$	$t_L(\lambda) + O(\ell + Ln\lambda)$	$O(\ell)$

the codeword size. The specific construction of distillation codes in [76] adds an additional overhead of a signature.

The second and third lines detail the cryptographic and encoding/decoding computational overhead *in addition to* the cost for encoding the input message. The sign-all approach requires a signature operation per code symbol, as does the NM construction. The latter also requires encrypting/decrypting each symbol. The list decoding schemes of [106] and [111] both perform a single signature operation when encoding, but require a list decoding step and one signature operation for each item in the decoded list. Our list decoding based scheme also requires the verification of n hashes corresponding to the symbols and requires an additional encoding step to encode the hashes. The Merkle tree constructions require $2n - 1$ hash operations to build the Merkle tree and n instances of $\log n$ hash operations to verify the received codeword.[5] Distillation codes also perform an additional signature operation.

As is clear from Tables 3.1 and 3.2, our constructions provide stronger guarantees of integrity than several previous works. Specifically, we are able to correct more errors than the sign-all and Merkle tree approaches when against an independently corrupting adversary. The additional overhead in our schemes while providing this increased security is small: n hash operations for LD and n PRP and encryption operations for NM. Moreover, in our LD scheme, when given fixed n and k, the overhead from list decoding is *constant* as the symbol size grows. Compare this to [106] where the list decoding time increases as symbol grows since the field operations become more expensive. Finally, it should be noted, that for some efficiently decodable list decoding algorithms, even though the maximum

[5]The $\log n$ verifications per block is necessary since we cannot simply reconstruct the whole tree and compare the root hashes as there may be corrupted symbols and hashes.

list size L is polynomially bounded, on average, the output list is very small (e.g., see the appendix in [109] analyzing the average list size of the Guruswami-Sudan decoder for Reed-Solomon codes).

3.3 Preliminaries

In this section we define the basic terms and concepts, both coding theoretic and cryptographic, that we will use throughout this work.

Notation. We denote the security parameter by λ. If Alg is a probabilistic algorithm, let $[\mathsf{Alg}(\pi)]$ denote the set of all possible outputs of Alg when run on parameters π. Let $x \xleftarrow{R} S$ denote sampling x from the set S uniformly at random. Let $x \leftarrow D$ denote sampling x according to distribution D. We also use $\leftarrow$ for assignment in algorithm listings, which will be clear from the context. We denote string concatenation with $\circ$. We abbreviate probabilistic polynomial time with PPT. We use $|S|$ to denote the cardinality of a set S and we let $P(n)$ denote the set of all permutations over $\{0,1\}^n$.

3.3.1 Coding Theory

An *error correcting code* (ECC) is an encoding of an input message into another form such that the encoded message can be transmitted over a noisy channel while still permitting recovery of the original message by the receiver. That is, an ECC transforms a message and adds some "redundancy" such that some (bounded) portion of the message may be corrupted (or lost entirely) but the recipient of the message can still reverse the encoding and recover the message.

An error correcting code C is defined over a set Σ, called the *alphabet*, elements of which are called *symbols*—often Σ is a finite field. The code C maps elements of Σ^k (called *messages*) to elements of Σ^n (called *codewords*), for some $n \geq k$. k is called the *message length* of the code C and n is the *block length*. For an element $(m_1, \ldots, m_k) \in \Sigma^k$, each m_i is called a *message symbol*; similarly, for $(c_1, \ldots, c_n) \in \Sigma^n$, each c_i is called a *code symbol* The ratio $R = k/n$ is called the *rate* of the code: roughly, this is the amount of information transmitted per codeword. The *Hamming distance* between two codewords x and y is defined as $\Delta(x, y) = |\{i \mid 1 \leq i \leq n, x_i \neq y_i\}|$. A code C has *minimum distance* d if for all codewords $x, y \in \Sigma^n$ such that $x \neq y$, we have that $\Delta(x, y) \geq d$. Note that $d \leq n - k + 1$; if $d = n - k + 1$, then C is called *maximum distance separable* code, or an *MDS code*. We use the notation $[n, k]$ to indicate a code of message length k and block length n.

Definition 3.1 (Error Correcting Code)**.** An *error correcting code* C over an alphabet Σ with minimum distance d, is a pair of maps $(\mathsf{Encode}, \mathsf{Decode})$, where $\mathsf{Encode} : \Sigma^k \to \Sigma^n$ and $\mathsf{Decode} : \Sigma^n \to \Sigma^k$, such that for all $m \in \Sigma^k$ and for all $c \in \Sigma^n$ such that $\Delta(c, \mathsf{Encode}(m)) \leq \lfloor d/2 \rfloor$, we have that $\mathsf{Decode}(c) = m$. □

In some situations, the decoder may fail to recover a message: we denote this by having the decoder Decode output $\perp$ and call this *decoding failure*. If Decode outputs a message other than the original message, we call this a *decoding error*.

For a code with minimum distance d, the code can be uniquely decoded for up to $\lfloor d/2 \rfloor$ errors (called the *unique decoding radius*). However, if we allow the decoder to output a *list* of possible decodings, such that the correct decoding is in that list, then it is possible to correct more than $\lfloor d/2 \rfloor$ errors. This is called *list decoding*, defined next.

Definition 3.2 ((ρ, L)-List-Decodability)**.** For $0 < \rho < 1$ and an integer $L \geq 1$, a code C with range $R \subset \Sigma^n$ is said to be list decodable up to a fraction ρ of errors with list size L, or (ρ, L)-*list-decodable*, if for every $y \in \Sigma^n$, the size of the set $\{c \mid c \in R, \Delta(c, y) \leq \rho n\}|$ is at most L. We say that C is *efficiently* (ρ, L)-*list-decodable* if C is (ρ, L)-list-decodable and there exists a PPT list decoding algorithm LD which, on input $x \in \Sigma^n$, outputs all codewords $c \in R$ such that $\Delta(x, c) \leq \rho n$. □

Another way of viewing an error correcting code is as an embedding of Σ^k into Σ^n such that the Hamming distance between any two codewords is (at least) the minimum distance d. Since the Hamming distance is a metric we can use it to define spheres around each codeword. The unique decoding radius is then the radius of the largest sphere that can be centered at each codeword without intersecting any other sphere. Thus any elements of Σ^n inside the unique decoding radius for a given codeword are unambiguously decoded to the corresponding message of that codeword. If we allow the radius to increase, then the spheres begin to overlap and decoding an element in intersection becomes ambiguous. List decoding an element $c \in \Sigma^n$ can then be viewed as producing a list of the centers of each of the ciphers containing c, where each center is an element in the range of Encode.

Reed-Solomon codes [138] are a particular ECC that have seen wide applicability and deployment (see [166] for examples). The codes themselves are conceptually simple: fundamentally, they are based on noisy polynomial interpolation. In particular, an input message is broken up into k symbols which are regarded as coefficients of a degree $k-1$ polynomial which is then evaluated at n points and these evaluations constitute the encoding. Some of the code symbols become corrupted in transit and

the decoder executes a particular polynomial interpolation algorithm on these "noisy" symbols to try and recover the original message. We present now the formal definition of a Reed-Solomon code.

Definition 3.3 (Reed-Solomon Code)**.** An $[n, k]$ Reed-Solomon code C over a finite field $\mathbb{F}$ is a map $\mathbb{F}^k \to \mathbb{F}^n$ where each element $a = (a_0, \ldots, a_{k-1}) \in \mathbb{F}^k$ is interpreted as a polynomial $p(x) = \sum_{i=0}^{k-1} a_i x^i \in \mathbb{F}[x]$, and then we define $\mathsf{Encode}(a) = (p(\alpha), p(\alpha^2), \ldots, p(\alpha^n))$, where α is a primitive root of $\mathbb{F}$. □

Note that the of use a primitive root is not required: any n *distinct* points $\alpha_1, \ldots, \alpha_n$ will suffice. Reed-Solomon codes are MDS codes and have efficient encoding and decoding algorithms, taking $O(nk)$ and $O(n^2)$ time, respectively.[6] Moreover, Reed-Solomon codes can be efficiently list decoded using the Guruswami-Sudan algorithm [109] or by using a variant called *folded Reed-Solomon codes*, described in [58]. The latter achieve the best possible list decoding for Reed-Solomon codes, being able to correct up to $n - k$ errors. We use Reed-Solomon codes in our experiments, but our constructions are not dependent upon them.

3.3.2 Cryptography

We use several basic tools from cryptography in this work and present their definitions here. First, we use a collision-resistant hash function in our list-decoding-based construction. Informally, a hash function is a function that takes an arbitrary sized input and produces a fixed sized output. Such a function is said to be *collision-resistant* if it is infeasible to find any two messages x and y that map to the same output. More formally, we have the following definition.

Definition 3.4 (Collistion-Resistant Hash Function)**.** A family of (efficiently computable) deterministic functions $\mathcal{H}$ (where for each $h \in \mathcal{H}$, $h : \{0,1\}^* \to \{0,1\}^\lambda$) is said to be (t, ε)-*collision-resistant* if for all PPT adversaries $\mathcal{A}$ running in time t,

$$P[h \xleftarrow{R} \mathcal{H}; \mathcal{A}(1^\lambda, h) \to x, y : x \neq y \wedge h(x) = h(y)] \leq \varepsilon$$

where the probability is taken over the random selection of h and the random coins of $\mathcal{A}$. □

If we relax the time restriction on $\mathcal{A}$, allowing it to run for any polynomial amount of time instead of just time t, then if ε is negligible, we call h *collision-resistant* or just *cryptographically secure*.

[6] Using discrete Fast Fourier Transforms, encoding and decoding can be performed in $O(n \log n)$ time. But, those algorithms are asymptotic. For practical values of k and n (i.e., in the 10 or 100s), using a combination of Horner's Rule for encoding and the Berlekamp-Massey algorithm [107] for decoding is generally more efficient.

Drawing the function h at random from a family of hash functions $\mathcal{H}$ prevents the degenerative case where we fix h before choosing the adversary $\mathcal{A}$, and then $\mathcal{A}$ comes "pre-programmed" with a specific collision for h, trivially winning the game. For simplicity, though, we will often just refer to a "collision-resistant hash function" and omit drawing it from a family of functions. In practice, there is typically just a single hash function h (for example, SHA256) that is used where, even though mathematically, there exists an adversary $\mathcal{A}$ that knows collisions in h, it is infeasible to find or construct such an adversary.

Our second authenticated error correcting code uses a *pseudorandom permutation* (PRP) in securing each symbol. Roughly, a PRP is a function taking two inputs $k \in \{0,1\}^\lambda$ and $x \in \{0,1\}^n$ and producing an output $f(k,x) \in \{0,1\}^n$. In particular, the function $f_k(\cdot) = f(k,\cdot)$ is a permutation on the set $\{0,1\}^n$ (and called a *keyed permutation*); moreover, for k chosen at random from $\{0,1\}^\lambda$, f_k is *indistinguishable* from a permutation selected at random from $\mathcal{P}(n)$. Let $\mathcal{O}_f$ denote an oracle for the permutation $f_k(\cdot)$ (where $k \leftarrow \{0,1\}^\lambda$). (We do not give the adversary $\mathcal{A}$ access to an oracle for f^{-1} and hence only require a "weak" PRP.)

Definition 3.5 (Pseudorandom Permutation)**.** A keyed permutation $f : \{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}^n$ is (t,q,ε)-*indistinguishable*, if for all PPT distinguishers $\mathcal{D}$ running in time t and making q queries to $\mathcal{O}_f$, distinguishes f from a permutation $\pi \xleftarrow{R} \mathcal{P}(n)$ with probability at most $\frac{1}{2} + \varepsilon$. □

If we relax the restrictions on $\mathcal{D}$ and only require that t and q be polynomially-bounded, then if ε is negligible, we say that f is *computationally indistinguishable* from a random permutation, or just *computationally secure*. Note that a (secure) block cipher is an example of a pseudorandom permutation.

Another cryptographic tool that we employ is a public-key signature scheme. A public-key signature scheme (or just a signature scheme) is triple of algorithms, consisting of a key generator, a signing function, and a verification function. A signature scheme allows a user to generate a digital signature for a document, akin to a physical signature, in such a way that it is (computationally) infeasible to forge. The signature is generated with a secret key, and the public key (related to the private key) is used to verify the signature. More formally, we have the following definition.

Definition 3.6 (Public-Key Signature Scheme)**.** A *public-key signature scheme* Π for a message space $\mathcal{M}$ is the triple of algorithms, $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ where Gen and Sign are PPT algorithms and Verify is a deterministic algorithm, such that,

- Gen: on input 1^λ, outputs a pair (pk, sk) where pk is the *public key* and sk is the *secret key* (or *private key*).
- Sign: on input secret key sk and $m \in \mathcal{M}$, outputs a signature σ.
- Verify: on input public key pk, $m \in \mathcal{M}$, and signature σ, outputs a bit $b \in \{0,1\}$.

We require that for all $(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda)$ and for all $m \in \mathcal{M}$, $\mathsf{Verify}(pk, m, \mathsf{Sign}(sk, m)) = 1$. □

The secret-key analog of a public-key signature scheme is a message authentication code (MAC).

Definition 3.7 (Message Authentication Code)**.** A *message authentication code* (MAC) for a message space $\mathcal{M}$ and tag space $\mathcal{T}$, is the triple of algorithms, $M = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Verify})$ where,

- Gen is a PPT algorithm that on input 1^λ outputs a key k.
- Mac is a PPT algorithm that on key k and $m \in \mathcal{M}$ outputs a *tag* $\tau \in \mathcal{T}$.
- $\mathsf{VerifyMac}$ is a deterministic algorithm that on input key k, $m \in \mathcal{M}$, and $\tau \in \mathcal{T}$ outputs a bit $b \in \{0,1\}$.

We require that for all $k \leftarrow \mathsf{Gen}(1^\lambda)$ and all $m \in \mathcal{M}$, $\mathsf{Verify}(k, m, \mathsf{Mac}(k, m)) = 1$. □

Finally, we also use public-key encryption schemes.[7] A public-key encryption scheme consists of a key generation algorithm, an encryption function, and a decryption function. Encryption is allowed to be probabilistic, but decryption must be deterministic. Some definitions allow the decryption to fail to decrypt a ciphertext with some small probability, but we will assume that decryption never fails when given a legitimate message as input. A secret-key encryption scheme is defined analogously by omitting the public key pk and having the encryption algorithm take sk as input.

Definition 3.8 (Public-key Encryption Scheme)**.** A *public-key encryption scheme* for a message space $\mathcal{M}$ (with associated ciphertext space $\mathcal{C}$), is the tuple of algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ where, Gen and Enc are PPT algorithms and Dec is a deterministic algorithm, such that,

- Gen: on input 1^λ, outputs a key (pk, sk) where pk is the *public key* and sk is the *secret key*.
- Enc: on input a public key pk and a message $m \in \mathcal{M}$, outputs a ciphertext $c \in \mathcal{C}$.
- Dec: on input a key sk and a ciphertext $c \in \mathcal{C}$, outputs a message $m \in \mathcal{M}$.

We require that for all $k \leftarrow \mathsf{Gen}(1^\lambda)$ and for all $m \in \mathcal{M}$, $\mathsf{Dec}(sk, \mathsf{Enc}(pk, m)) = m$. □

[7] Nothing in our constructions and proofs require asymmetric keys. Using secret-key variants (i.e., symmetric encryption and MACs) is a straightforward optimization of our scheme. Indeed, we use these in our experiments in Section 3.7.

3.3.3 Security Definitions

For each of the cryptographic tools defined above, we present here their corresponding security definitions. The definitions presented are concrete in that they specify exactly the resources available to the attacker to break the scheme. This formulation allows for a precise characterization of the efficiency our reductions in our security proofs and provides guidance for parameterizing a scheme to achieve a desired level of security. Though, for simplicity of presentation we will often omit the exact security of a given primitive.

Basic Definitions. First, for digital signatures, intuitively, we would like digital signatures to be unforgeable, similar to the "guarantee" of real-world signatures. The strongest notion of unforgeability is that no adversary can forge a signature for *any* message, even one of its own choosing. In the below definition, the adversary $\mathcal{A}$ is trying to win a game. In particular, we give $\mathcal{A}$ access to a signature oracle $\mathcal{O}_{sk}$ that on input m produces a signature σ for m. After some number of queries q, $\mathcal{A}$ outputs a message and signature pair (m', σ') and wins the game if $\mathsf{Verify}(sk, m', \sigma') = 1$. We also require that m was not previously queried to the oracle, otherwise $\mathcal{A}$ can trivially win the game.

Definition 3.9 (Unforgeable Signature Scheme)**.** A public-key signature scheme Π is (t, q, ε)-*unforgeable* if for all PPT adversaries $\mathcal{A}$ running in time t and given access to an oracle $\mathcal{O}_{sk}$,

$$P[(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda); \mathcal{A}^{\mathcal{O}_{sk}(\cdot)}(1^\lambda, pk) \rightarrow (Q, m, \sigma) : m \notin Q \wedge \mathsf{Verify}(pk, m, \sigma) = 1] \leq \varepsilon$$

where Q is the list of queries made to $\mathcal{O}_{sk}$, such that $|Q| \leq q$, and the probability is taken over the random coins of Gen, $\mathcal{A}$ and $\mathcal{O}_{sk}$. $\square$

Relaxing the restrictions on t and q and allowing them to just be polynomially-bounded instead of fixed, if ε negligible in λ, then we call the signature scheme *existentially unforgeable*. The definitions of (t, q, ε)-unforgeability and existential unforgeability for MACs are analogous and can be constructed by simply omitting the public key and using the secret key in $\mathsf{VerifyMac}$.

An encryption scheme (either public-key or secret-key) is *semantically secure* when each ciphertext hides all of the information about its message in the following sense: any function f of the message that can be computed with the ciphertext can also be (efficiently) computed *without* it. Intuitively, the ciphertext is not "helpful" to $\mathcal{A}$ when computing $f(m)$. To prevent $\mathcal{A}$ from trivially knowing the plaintext beforehand, we sample the message m from a distribution $\mathcal{D}$ over the message space

$\mathcal{M}$. The distribution $\mathcal{D}$ must be *efficiently-samplable*: i.e., there exists an efficient (polynomial-time) algorithm S that on input 1^λ outputs a message $m \in \mathcal{M}$ according to distribution $\mathcal{D}$. $\mathcal{A}$ may have some auxiliary information about m that it knows or can learn, e.g., m is an encrypted network packet with a specific format: we denote this auxiliary information by the function h (the "history" of m).

Definition 3.10 (Semantic Security)**.** A public-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is (t, o, ε)-*semantically-secure* if for every efficiently-samplable distribution $\mathcal{D}$ over message space $\mathcal{M}$, all functions $h : \mathcal{M} \to \{0,1\}^*$ and $f : \mathcal{M} \to \{0,1\}^*$ (of arbitrary complexity) and every algorithm $\mathcal{A}$ running in time t, there is an algorithm $\hat{\mathcal{A}}$ that runs in time $\leq t + o$ such that,

$$|P[m \leftarrow \mathcal{D};\ (pk, sk) \leftarrow \mathsf{Gen}(1^\lambda);\ \mathcal{A}(1^\lambda, pk, \mathsf{Enc}(pk, m), h(m)) = f(m)] \\ -P[m \leftarrow \mathcal{D}; \hat{\mathcal{A}}(1^\lambda, h(m)) = f(m)]| \leq \varepsilon,$$

where the probabilities are taken over the random coins of $\mathcal{D}$, Gen, Enc, $\mathcal{A}$, and $\hat{\mathcal{A}}$.[8]

As an example, a pseudorandom permutation is semantically secure since, intuitively, the encryption $f(k, m)$ for a random k and a given m is indistinguishable from a random value (i.e., the output of a truly random permutation). Hence, the ciphertext is "unrelated" to the input message, and so it is not "helpful" for computing any function of m.

Non-malleability. A strong security property that a cipher can have is one of *non-malleability*, which, at a high-level, means that an adversary cannot manipulates a ciphertext and have the resulting plaintext possess a desired property (e.g., have a particular formatting). More formally, for any polynomial-time computable relation $\mathcal{R}$, given an encryption of a message m, the adversary cannot produce a ciphertext c' such that with $m' = \mathsf{Dec}(sk, c')$, $\mathcal{R}(m, m'))$ holds with significant advantage over simply selecting m' at random. This idea was first formalized in [36]. One advantage of proving that a cipher is non-malleable is that non-malleability combined with CPA-security implies that the cipher is CCA2-secure, the strongest form of security for a cipher (see [36]). We use the formalization given in [12] adapted to be a concrete formulation. In the following, M indicates a set of messages. M is *valid* if all messages with non-zero probability are the same length (since encryption is not meant to hide the length of the plaintext).

[8] Parameter o is $\hat{\mathcal{A}}$'s overhead required to produce the same output as $\mathcal{A}$.

Definition 3.11 (Non-malleable Cipher)**.** Let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a public-key encryption scheme, let $\mathcal{R}$ be a relation, let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ be an adversary consisting of a pair of algorithms and let $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$ be a pair of algorithms that we call the simulator. For atk $\in$ {cpa, cca1, cca2} and λ the security parameter, define,

$$\mathsf{Adv}^{\text{snm-atk}}_{\mathcal{A},\mathcal{S},\Pi,\mathcal{R}}(\lambda) = |\Pr[\mathsf{Expt}^{\text{snm-atk}}_{\mathcal{A},\Pi,\mathcal{R}}(\lambda) = 1] - \Pr[\mathsf{Expt}^{\text{snm-atk}}_{\mathcal{S},\Pi,\mathcal{R}}(\lambda) = 1]|,$$

where,

$\mathsf{Expt}^{\text{snm-atk}}_{\mathcal{A},\Pi,\mathcal{R}}(\lambda)$
$(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda)$
$(M, s_1, s_2) \leftarrow \mathcal{A}_1^{\mathcal{O}_1^{\mathsf{Dec}}}(1^\lambda, pk)$
$x \xleftarrow{R} M;\ y \leftarrow \mathsf{Enc}(pk, x)$
$\vec{y} \leftarrow \mathcal{A}_2^{\mathcal{O}_2^{\mathsf{Dec}}}(s_2, y)$
$\vec{x} \leftarrow \mathsf{Dec}(sk, \vec{y})$
If $\mathcal{R}(x, \vec{x}, M, s_1)$ then return 1
Else return 0

$\mathsf{Expt}^{\text{snm-atk}}_{\mathcal{S},\Pi,\mathcal{R}}(\lambda)$
$(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda)$
$(M, s_1, s_2) \leftarrow \mathcal{S}_1(1^\lambda, pk)$
$x \xleftarrow{R} M$
$\vec{y} \leftarrow \mathcal{S}_2(s_2)$
$\vec{x} \leftarrow \mathsf{Dec}(sk, \vec{y})$
If $\mathcal{R}(x, \vec{x}, M, s_1)$ then return 1
Else return 0

where,

- If atk = cpa, then $\mathcal{O}_1^{\mathsf{Dec}}(\cdot) = \varepsilon$ and $\mathcal{O}_2^{\mathsf{Dec}}(\cdot) = \varepsilon$
- If atk = cca1, then $\mathcal{O}_1^{\mathsf{Dec}}(\cdot) = \mathsf{Dec}(sk, \cdot)$ and $\mathcal{O}_2^{\mathsf{Dec}}(\cdot) = \varepsilon$
- If atk = cca2, then $\mathcal{O}_1^{\mathsf{Dec}}(\cdot) = \mathsf{Dec}(sk, \cdot)$ and $\mathcal{O}_2^{\mathsf{Dec}}(\cdot) = \mathsf{Dec}(sk, \cdot)$

We only consider adversaries $\mathcal{A}$ that are *legitimate* in the sense that with probability 1 the following are true in the first experiment: (1) the message space M is valid, (2) $y \notin \vec{y}$, and (3) in the case of CCA2, $\mathcal{A}_2$ does not query $\mathcal{O}_2$ with y. We only consider simulators that are legitimate in the sense that the message space M in the second experiment is valid. We say that Π is $(t, q_1, q_2, \varepsilon)$-*non-malleable* if for every $\mathcal{R}$ computable in time t, every (legitimate) $\mathcal{A}$ that runs in time t, with $\mathcal{A}_1$ making at most q_1 queries to $\mathcal{O}_1^{\mathsf{Dec}}$ and $\mathcal{A}_2$ making at most q_2 queries to $\mathcal{O}_2^{\mathsf{Dec}}$, and $\mathcal{A}$ and outputs a message space M samplable in time t, there exists a (legitimate) polynomial-time simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$ such that $\mathsf{Adv}^{\text{snm-atk}}_{\mathcal{A},\mathcal{S},\Pi,\mathcal{R}}(\cdot) \leq \varepsilon$. □

If Π is $(t, q_1, q_2, \varepsilon)$-non-malleable for all polynomially bounded t, q_1, and q_2 such that ε is negligible in λ, then we call Π simply *non-malleable-secure*. We note here that there is a subtlety in the security of non-malleable ciphers. The authors of [122] observe that if the adversary is allowed to output

invalid ciphertexts[9] then the definition above does *not* imply CCA2-security since the simulator may not be able to efficiently produce an invalid ciphertext without access to the decryption oracle. For simplicity, we restrict ourselves to consider adversaries that do not produce invalid ciphertexts. This is not restrictive in our case since we instantiate the non-malleable cipher in our scheme with a block cipher in CMC mode [60], in which all ciphertexts are valid.

3.4 Security Model

In this section we first motivate our new adversarial model and how it relates to securing distributed storage, and then we outline our model and present our new definitions. In Section 3.6 we prove our schemes secure in this model.

3.4.1 Motivation

For our security model, we first assume that the channel is computationally-bounded, as in [87] and [111]. We first note that previous work that combined cryptography and error correcting codes (to achieve better error correction) typically worked by simply appending an authentication tag or checksum to each code symbol (see [76] and [87]). We call this *authenticated error correcting codes*. There is an implicit assumption in each of these constructions that the channel would treat a symbol and its tag as a *single unit* instead of distinct pieces. If, however, we allow the adversary to corrupt the tag and the symbol *separately*, then due to the ambiguity inherent in tag verification failure (i.e., it could be a bad symbol or a bad tag), the additional error correction provided by using cryptography can be nullified (as detailed previously). Moreover, the authentication tag for a symbol constitutes a piece of *metadata* about that symbol and logically should be treated distinctly. And, indeed, the tag may not always be stored with the symbol itself.[10]

Indeed, several real-world systems store these checksums separate from the data. For example, ZFS stores its data in a tree of blocks, with data blocks in the leaves and the checksum of each block stored in its parent block (with the exception of the root, or über, block) [19]. The Hadoop File System (HDFS) stores the checksums for a file block in a separate file on the same host, but this does

[9] That is, if the set of possible messages for which ciphertexts can be efficiently created is *different* than the range of the decryption function.

[10] It must be noted, though, that often this information is stored with the data. Object stores, such as Amazon S3 [3] and Microsoft's Azure [46], store metadata with the data itself. The WAFL file system from NetApp [64] stores a 64-byte checksum with each 4KB disk block.

not guarantee that checksum will be physically located near (i.e., on the same disk as) the data block itself [134]. An extension of HDFS, called HDFS-RAID, encodes each file using a Reed-Solomon code with a checksum file stored on the same host as its corresponding symbol as before [135]. While HDFS normally stores multiple replicas of blocks for fault-tolerance, HDFS-RAID only stores a single copy of each code symbol (and its checksums), since the encoding itself will provide the necessary redundancy. This separation of data and authentication tags makes independent corruption of symbols and tags a real possibility. Thus, in this section we define an adversary who is capable of corrupting the code symbols and their associated authentication tags *independently* while previous work only considered adversaries that corrupt both. In Section 3.5 we will provide two constructions designed to withstand this adversary, and in Section 3.6 we will prove their security in this model.

The example of HDFS-RAID provides a strong case for why such an adversary should be addressed. In particular, HDFS-RAID (as well as HDFS in general) is capable of repairing an encoded file when part of it is lost. Suppose the file is encoded with an $[n, k]$ error correcting code and one of the code symbols is corrupted (e.g., due to a faulty disk). To repair the file, HDFS-RAID must read-in k of the $n-1$ remaining pieces (each of which is tens to hundreds of MB in size), decode the file and then re-encode the file to generate the missing symbol.[11] But, there is an inherent ambiguity whenever a checksum verification fails: either the data or the checksum may be corrupted (or both). Thus, the independent corruption of a checksum can lead to entirely *unnecessary* and *expensive* repair operations.

3.4.2 The Model

In our model, as stated before, we allow the code symbols and their associated authentication tags to be corrupted *independently*. Of course, the corruption of the encoding and its authentication tags must be bounded, otherwise an adversary can trivially destroy all of the data. As an example, say we encode a message with an ECC, sign each symbol with a secure signature scheme, and each signature is appended to its respective symbol. In our model, a ρ-fraction of symbols can be corrupted by the channel and, in addition, a ρ-fraction of tags can be corrupted independently of the corruption of the code symbols. In the remainder of this section, we will first define a public-key coding scheme, then define a computationally-bounded channel that attacks a public-key coding scheme, and finally we

[11] The large cost of these repair operations has been noted previously and novel error correcting codes were devised to reduce the cost, see [143]. But, even in that work, the repair operations are expensive.

define our "independently corrupting" adversary described above. Throughout this section we will use to phrase "corrupt a symbol" to include actual corruption of data, erasing the symbol, and even deleting a symbol.

Public-Key Coding Scheme. We base our definition of a public-key coding scheme on the definition of such a scheme in [111], but we adapt their formalization to be more concrete rather than asymptotic. For example, they define $\lambda = f(k, \epsilon) = \lfloor k^{\epsilon} \rfloor$, where ϵ is their security parameter, while in our definition, we have λ be independent of the message length k.

To ensure that all algorithms run in polynomial-time, we only consider *feasible* message sizes, i.e., where k is polynomially-bounded. Moreover, we only consider *non-negligible* code rates, i.e., $R > 1/p(\lambda)$ for some polynomial $p(x)$. This ensures that the values of n and k used in the code are always polynomially-bounded and so encoding and decoding always take polynomial time. Also, our formulation implicitly places a lower bound on the input size, as it is not possible to achieve security with parameter λ if the space of inputs is polynomial in λ. Hence, we require that the space of message (i.e., Σ^k) be superpolynomial in λ—for example, by setting $\Sigma = \{0,1\}^{\lambda}$.

We allow a channel to have a "memory" and save past messages sent over the channel and possibly use that information to manipulate future messages. Such channels are called *stateful*. If the channel keeps no state between message transmissions, it is called *memoryless*.

Definition 3.12 (Public-Key Coding Scheme)**.** A *public-key coding scheme* $\mathcal{CS}$ consists of two PPT algorithms Gen and Encode, a deterministic, polynomial-time algorithm Decode, a finite alphabet Σ, the *security parameter* λ and a *code rate* $0 < R < 1$: $\mathcal{CS} = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{Decode}, \Sigma, \lambda, R)$.

For all message lengths k and letting $n = \lfloor k/R \rfloor$,

- Gen: on input 1^{λ}, outputs a key-pair (pk, sk), where pk is the *public key* and sk the *secret key*.
- Encode: on input (1) an integer i less than 2^{λ}, the *counter*; (2) the secret key sk; and (3) a string $m_i \in \Sigma^k$, called the *message*, outputs $i+1$ (as the new counter) and a *codeword* $c_i \in \Sigma^n$, referred to as an *encoding* of m_i.
- Decode: on input the public key pk and an element $c_i' \in \Sigma^n$ (a possibly corrupted version of an encoding of a message m_i), outputs a decoded message $m_i' \in \Sigma^k \cup \{\perp\}$ (where $\perp$ indicates Decode failed to recover any message).

We require that for all $(pk, sk) \in [\mathsf{Gen}(1^{\lambda})]$, for all $m \in \Sigma^k$, and for all counters $i \in \mathbb{Z}_{2^{\lambda}}$, we have that $\mathsf{Decode}(pk, \mathsf{Encode}(sk, i, m)) = m$. $\square$

A *secret-key coding scheme* can be defined analogously and we define a *keyed coding scheme* $\mathcal{CS}$ to be either a secret-key coding scheme or a public-key coding scheme.

Note that the counter in the definition above is not strictly necessary: its purpose is to prevent replays of old messages. When sending over a memoryless channel, the counter can be omitted. An alternative way to prevent replays would be to use a pre-arranged sequence of one-time signatures. The sender, when encoding the next message, uses the next signature in the sequence to sign the message. The receiver, upon receiving a message, attempts to verify the signature using the next signature in the sequence and discards the public key of the signature regardless of whether or not the signature verification succeeds. Though, if a message is lost or of the adversary injects a spurious message when the sender sent none, the sender and receiver will become desynchronized with no way to recover other than by communicating out-of-band.

Computationally-bounded Channel. For a computationally-bounded channel, again, we model our definition on the definition found in [111], and, as before, we modify their definition to be more concrete. Intuitively, a computationally-bounded channel is a PPT algorithm $\mathcal{A}$ that takes a codeword c as input and outputs an element $c' \in \Sigma^n$, where c' is possibly a corrupted version of c. $\mathcal{A}$ *succeeds* when Decode, on input c', outputs a decoded message m' that is *different* from the originally encoded message m.

Previous models of computationally-bounded channels includes the work of Lysyanskaya et al. [106] where the authors define an (α, β)-network where for group of n packets sent in the network, an α-fraction survive transmission unscathed and at most βn packets arrive at the receiver (β is denial-of-service tolerance parameter). In their constructions, they use error correcting codes, signatures, and cryptographic hashes to authenticate the packets. The α and β parameters give implicit bounds on the number of code symbols that may be corrupted by the network. A motivating example for their model is an adversary that has partial (but not full) control of a network. In our model, we do not consider injection of extra symbols (though, it could easily be handled) and we allow the adversary to see all of the code symbols. That is, we allow the adversary controls the entire network but we place explicit restrictions on the amount of corruption he may inflict (which is standard in coding theory).

The work of Bowers et al. [21] defines an *adversarial error correcting code* where the code can tolerate some amount of adversarial corruption. They say that a particular adversarial error

correcting code is (β, δ)-bounded if, for an $[n, k]$ code, and a computationally-bounded adversary $\mathcal{A}$, $P[(c, c') \leftarrow \mathcal{A}(1^\lambda); \Delta(c, c') \leq \beta n] - \delta$ is negligible in λ. That is, the probability that the $\mathcal{A}$ causes two codewords c and c' to decode to different messages, where the distance between c and c' is less than βn, is negligibly different from some probability δ. This definition was created to model adversarial behavior against proofs-of-retrievability (PoRs). In a PoR, the construction provides two guarantees:1. if the corruption is below some threshold t, it can be repaired; and 2. if the corruption is above the threshold t, it will be detected with high probability. This then motivates the adversary $\mathcal{A}$ to minimize $\Delta(c, c')$ (while the code designer wants to maximize β). This definition, however, allows for the possibility that if $\Delta(c, c') > \beta n$ and $\Delta(c, c') < d$ (where d is the minimum distance of the code), then $\mathcal{A}$ can easily cause the codewords to decode to different messages. (That is, the definition allows for a *gap* in the error correction capacity of the underlying code and the tolerance to adversarial corruption.) In a PoR, this does not matter since the corruption (presumably) would be detected, but we are concerned with preserving the integrity of encoded messages regardless of the amount of corruption (up to the error correcting capacity of the code). We define our channel next.

Definition 3.13 (Computationally-Bounded Adversarial Channels, Attacks, and Successes)**.** A *computationally-bounded adversarial channel* $\mathcal{A}$ is a PPT algorithm that takes as input some state ψ and a codeword $c = \mathsf{Encode}(pk, i, m)$, for some $m \in \Sigma^k$ and $i \in \mathbb{Z}_{2^\lambda}$, and outputs an element $c' \in \Sigma^n$ and some new state ϕ' to be used against subsequent messages.

An *attack* of a computationally-bounded adversarial channel $\mathcal{A}$ on the public-key coding scheme $\mathcal{CS} = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{Decode}, \Sigma, \lambda, R)$ (with message length k) consists of the following process:

1. $\mathsf{Gen}(1^\lambda)$ outputs a key-pair (pk, sk).
2. $\mathcal{A}$ is given 1^λ and pk as input.
3. $\mathcal{A}$ outputs the first message m_1 to be transmitted, together with the state information ψ_1.
4. Encode then produces an encoding $c_1 \leftarrow \mathsf{Encode}(sk, 1, m_1)$
5. $\mathcal{A}$ is given ψ_1 and c_1 and computes (c_1', m_2, ψ_2), the received codeword c_1', the next message m_2, and the next state information ψ_2.
6. This process continues for some number of rounds, where at round $i + 1$, we have that $(c_i', m_{i+1}, \psi_{i+1}) \leftarrow \mathcal{A}(\psi_i, c_i)$ and $c_i \leftarrow \mathsf{Encode}(sk, i, m_i)$.

We denote the above experiment by $\mathsf{ChannelExp}_{\mathcal{CS},\mathcal{A}}(\lambda)$. For positive real $\rho < 1$, we say that the attack is *ρ-successful* if, from some stage i, (1) the Hamming distance between c_i and c_i' is at most ρn, and (2) the receiver makes a decoding error, that is, $\mathsf{Decode}(sk, r_i) \neq m_i$ and $m_i \neq \perp$. $\square$

An adversarial channel for secret-key coding schemes can be defined analogously. For a *memoryless* channel, there is only a single round of message selection, encoding, and corruption. The restriction that the decoder must output a message and not fail (and output $\perp$) is to ensure that the adversary does not win the game trivially. For $\rho < 1$ such that $\rho n > n - k$, $\mathcal{A}$ can win by simply erasing ρn symbols as this leaves less than k code symbols remaining, which, by the definition of an $[n, k]$ code, makes decoding impossible. Such attacks are denial-of-service attacks which are always possible, so we require that $\mathcal{A}$'s attack must be "useful." We now define a computationally secure coding scheme.

Definition 3.14 (Computationally Secure Coding Scheme)**.** A public-key (resp. secret-key) coding scheme $\mathcal{CS} = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{Decode}, \Sigma, \lambda, R)$ is (t, q, ε)-*computationally-secure against an error rate* ρ if, for all computationally-bounded adversarial channels $\mathcal{A}$ running in time t and participating in at most q rounds in $\mathsf{ChannelExp}_{\mathcal{CS},\mathcal{A}}(\lambda)$, for all (sufficiently large) message lengths k, and every $e \leq \rho$, the probability that an attack by $\mathcal{A}$ on $\mathcal{CS}$ with message length k is e-successful is at most ε. □

Independently-corrupting Adversary. Finally, we define our new adversary who corrupts the symbol-tag pairs in a way that is more powerful than previously considered. As stated before, we allow the channel $\mathcal{A}$ to apply its error rate ρ to the symbols and the authentication tags independently of each other. For example, if $\mathcal{A}$ corrupts m symbols, then it may also corrupt m tags without regard to which symbols were corrupted.

Definition 3.15. Let $\mathcal{A}$ be a computationally-bounded, adversarial channel with error rate ρ. And let $C = (\mathsf{Encode}, \mathsf{Decode})$ be an error correcting code with message length k, block length n, and alphabet Σ, with each code symbol σ authenticated by some tag τ produced by tagging scheme T. We say that $\mathcal{A}$ is *jointly-corrupting* if for any message $m \in \Sigma^k$, for each symbol-tag pair (σ, τ) output by Encode, $\mathcal{A}$ corrupts both σ and τ or neither (up to a ρ-fraction of all pairs). Otherwise we say that $\mathcal{A}$ is *independently-corrupting*. □

Put another way, suppose we have an $[n, k]$ error correcting code C where each symbol is authenticated with a MAC. Further, suppose $\mathcal{A}$ corrupts a set of at most m symbols and a set of at most m MACs. Let S_1 be the set of indices of corrupted symbols and S_2 to be the set of indices of corrupted MACs. If $\mathcal{A}$ is jointly-corrupting, then we require that $S_1 = S_2$ and that $|S_1|, |S_2| \leq m$. While, if $\mathcal{A}$ is independently-corrupting, then we allow $S_1 \neq S_2$, and have $|S_1|, |S_2| \leq m$. Previous

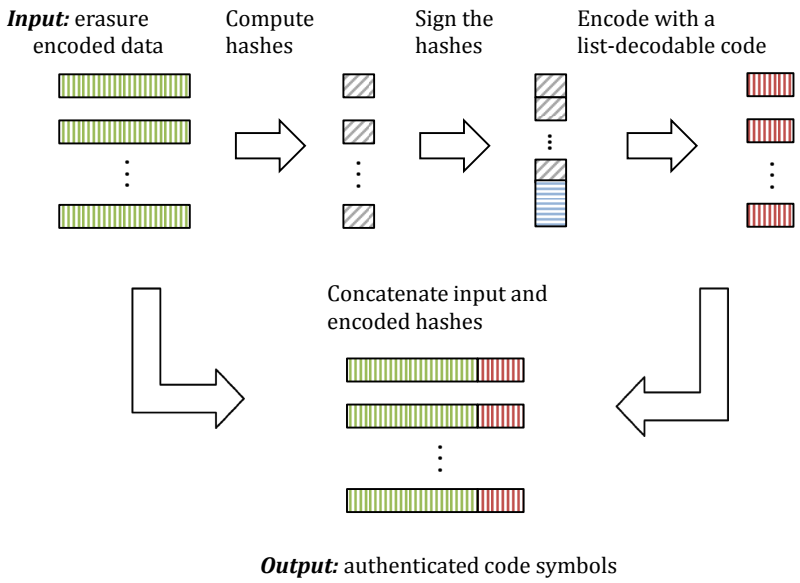


Figure 3.1: Diagram of the encoding function for AuthECC-LD.

constructions implicitly used a jointly-corrupting adversary with error rate ρ or an independently-corrupting adversary with error rate $\rho/2$. Our constructions are secure against *both* jointly-corrupting and independently-corrupting adversaries with error rate ρ.

3.5 Constructions

In this section we present two constructions of *authenticated error correcting codes*. Our constructions allow one to transform an erasure code into an error correcting code over a computationally-bounded channel with a small decrease in the code rate. When applied to an ECC, the constructions allow us to push the error correcting capacity of the code to be equal to (or very close to) that of the erasure correcting capacity.

3.5.1 List-decodable AuthECC

Informally, in our first construction, we encode a file using an erasure (or error correcting) code and compute an authentication tag for each code symbol. We then sign the concatenation of the tags, append the signature, and then encode the combined tags and signature with an efficiently list-decodable ECC. When decoding, we first list decode the encoded authentication tags and use

Algorithm 3.1 AuthECC-LD Encode: The encoding function for the list decoding based authenticated ECC with minimum distance d.

Input: Security parameter 1^λ, key pair (pk, sk), counter ℓ, message $m \in \Sigma^k$
Output: Encoded message $c \in \tilde{\Sigma}^n$

1: Set $(c_1, \ldots, c_n) \leftarrow \mathsf{Encode}(n, k, m)$
2: Set $h_i \leftarrow h(c_i)$ for $1 \leq i \leq n$
3: Set $\sigma \leftarrow \mathsf{Sign}(sk, h_1 \circ \ldots \circ h_n \circ \ell)$
4: Set $m' \leftarrow h_1 \circ \ldots \circ h_n \circ \ell \circ \sigma$ ▷ Gather tags, nonce, and signature for encoding
5: Set $(c'_1, \ldots, c'_n) \leftarrow \mathsf{EncodeLD}(n, k, m')$ ▷ EncodeLD has minimum distance d
6: Output $((c_1, c'_1), \ldots, (c_n, c'_n))$

the signature to identify the correct codeword in the list.[12] We then use the tags to sieve out the corrupted code symbols, marking them as erasures, and then using the erasure decoding algorithm on the remaining symbols and recover the message. Fundamentally, this construction decouples the error pattern in the authentication information from the error pattern in the message so that they can be adversarially corrupted independently of each other. Since this construction is an authenticated error correcting code based on list decoding, we call it AuthECC-LD.

Let C be an $[n, k]$ erasure code, where the message length is k, the codeword (or block) length is n, and the alphabet is Σ.[13] Let Encode and Decode be the erasure encoding and decoding functions and let $\mathsf{EncodeLD}$ and $\mathsf{DecodeLD}$ be the encoding and decoding functions for the list-decodable code. Let $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ be a signature scheme, and let s denote the length in bits of a signature generated by Π. Let $h : \Sigma \to \{0,1\}^\lambda$ be a collision-resistant hash function that takes as input a symbol from Σ and produces a value in $\{0,1\}^\lambda$. We assume that the length of the authentication tags and signature, $n\lambda + s$, is divisible by k so that we do not have any fractional symbols when encoding with the list-decodable code. We construct an $[n, k]$ error-correcting code $\tilde{C}$ over the alphabet $\tilde{\Sigma} = \Sigma \times \Sigma_l$, where $\Sigma_l = \{0,1\}^l$ and $l = (n\lambda + s)/k$.

Note that the security of the hash function puts a lower-bound on the size of the input message: i.e., to provide $\lambda/2$-bits of security, the input to the function must be at least λ-bits in size. While this precludes the possibility of using arbitrary message alphabets (i.e., symbols less than λ-bits in size), our primary application of this scheme is in secure storage where symbols sizes are large.

The encoding algorithm is depicted in Figure 3.1 and specified in Algorithm 3.1; the decoding algorithm is specified in Algorithm 3.2. Briefly, we encode the input message m with an $[n, k]$ erasure

[12] Using a signature to disambiguate list decoding was first employed in [106] and then independently discovered (and more formalized) in [111].

[13] Often in the literature for ECCs, $[n, k]$ will be used to denote a *linear* ECC (such as a Reed-Solomon code) while (n, k) is used for any ECC (linear or not). We use the former since the distinction is not important in this work.

Algorithm 3.2 AuthECC-LD Decode: The decoding function for the list decoding based authenticated ECC with minimum distance d.

Input: Security parameter 1^λ, public key pk, counter ℓ, codeword $c \in \tilde{\Sigma}^n$
Output: Message $m \in \Sigma^k$ or $\perp$

Parse $c = ((c_1, c'_1), \ldots, (c_n, c'_n))$ and set $c' = (c'_1, \ldots, c'_n)$ and $c_{msg} = (c_1, \ldots, c_n)$
Set $L \leftarrow \mathsf{DecodeLD}(n, k, c')$ ▷ DecodeLD has minimum distance d
Set $auth = \perp$
for each codeword $\hat{c}$ in L **do** ▷ Search the list for the correct decoded message
    Parse $\hat{c} = h_1 \circ \ldots \circ h_n \circ \ell' \circ \sigma$
    if $\mathsf{Verify}(pk, h_1, \circ \ldots \circ h_n \circ \ell', \sigma) = 0$ or $\ell' \neq \ell$ **then** ▷ Discard: invalid authentication info
        Discard $\hat{c}$
    else
        if $auth = \perp$ **then**
            Set $auth \leftarrow h_1 \circ \ldots \circ h_n$ ▷ Valid signature, save the value
        else
            Output $\perp$ and exit ▷ Two valid signatures: fail
        end if
    end if
end for
if $auth = \perp$ **then**
    Output $\perp$ and exit ▷ No valid signature or authentication tags
end if
Break $auth$ into $h_1, \ldots, h_n$
for each c_i in c_{msg} **do** ▷ Sieve out corrupted symbols
    if $h(c_i) \neq h_i$ **then**
        Set $c_i \leftarrow \perp$ in c_{msg}
    end if
end for
Output $\mathsf{Decode}(n, k, c_{msg})$ ▷ If too many c_i were erased, then Decode fails

code using Encode to produce a codeword c. We then hash each symbol of c using h, concatenate the hashes $h_1, \ldots, h_n$, and then sign the concatenation with Π to produce a signature σ. We create the authentication information m' by appending the signature to the hashes and apply an efficiently list-decodable code $\mathcal{L}$ to m' to create the codeword c'. Each symbol from c' is appended to a symbol in c and the resulting pairs are transmitted. The decoding function is largely the reverse of the encoding function. First, it list decodes the encoded tags and signature, checking the signature on each entry in the list until one validates.[14] If more than one signature validates (unlikely), the algorithm fails. The hashes in the decoded authentication information are then used to sieve out the corrupt code symbols. If too many symbols are corrupted, or if none of the signatures are valid, then decoding again fails. Else, decoding finishes by erasure decoding the remaining good pieces.

[14] Technically, DecodeLD outputs a list of *codewords* each of which must be decoded to recover the candidate authentication information before it can be verified. For simplicity, we omit this step from Algorithm 3.2 as it is not essential to understand the algorithm.

Efficiency. Our addition of authentication information adds some amount of computational overhead to the encoding and decoding of the erasure code. For the encoder, we must perform n hashes over elements of size $\log q$ (where q is the size of the alphabet Σ): denote this time by t_h. Let $t_e(C)$ and $t_d(C)$ denote the time to encode and decode (resp.) the erasure code C. The hashes are signed (taking time t_σ) and encoded using the list-decodable code $\mathcal{L}$ over an alphabet of size 2^l where $l = (s+nh)/k$ and s and h the lengths of the signature and a hash (resp.), taking time $t_e(\mathcal{L})$. Decoding first requires list decoding the authentication information, taking time $t_L(\mathcal{L}, \rho)$ to produce a list of size at most L for code $\mathcal{L}$ with error rate ρ. The decoder then performs at most L signature verifications to sieve out the spurious list items. This is followed by at most n hashing operations to filter out the corrupted erasure encoded symbols, and then finally the erasure code is decoded, giving us the following lemma.

Lemma 3.1. *Let* AuthECC-LD *be instantiated with an $[n, k]$ erasure code C over alphabet Σ, where $|\Sigma| = q$, and utilize a (ρ, L)-list-decodable code $\mathcal{L}$ (with minimum distance d over alphabet Σ' where $|\Sigma'| = 2^l$ and $l = (s+nh)/k$). Then, the encoder for* AuthECC-LD *takes time $t_e(C)+t_e(\mathcal{L})+O(nt_h+t_\sigma)$ and the decoder takes time $t_d(C) + t_L(\mathcal{L}, \rho) + O(nt_h + Lt_\sigma)$ for error rate ρ where $\rho n < d$.*

We note that often, a hash function takes time that is linear in the input size, such as SHA256; and, hence, in practical instantiations, we have $t_h = O(\log q)$. Similarly, signature schemes typically hash the input message of length l, produce a fixed-sized hash of length $l' = O(\lambda)$, and then sign the hash, giving a running time of $t_s = O(l)$ since l' is constant. Also, hash function and signatures can have varying lengths depending on the scheme and the parameterization but in each case the length is at least as long as the security parameter λ. For simplicity, we can denote the output length of both hash functions and signatures by λ. With these in mind, we get the following corollary.

Corollary 3.1. *Let* AuthECC-LD *be instantiated with an $[n, k]$ erasure code C over alphabet Σ, where $|\Sigma| = q$, and utilize a (ρ, L)-list-decodable code $\mathcal{L}$ (with minimum distance d over alphabet Σ' where $|\Sigma'| = 2^l$ and $l = (\lambda + n\lambda)/k$). Then, the encoder for* AuthECC-LD *takes time $t_e(C) + t_e(\mathcal{L}) + O(n(\log q + \lambda))$ and the decoder takes time $t_d(C) + t_d(\mathcal{L}, \rho) + O(n(\log q + L\lambda))$ for error rate ρ where $\rho n < d$.*

Decrease in Code Rate. The list-decodable code $\mathcal{L}$ is a code from $\Sigma_l^k \to \Sigma_l^n$ (recall that $\Sigma_l = \{0,1\}^l$, where $l = (n\lambda + s)/k$, λ is the length of a hash, and s is the length of a signature). Let b be the length (in bits) of an element of Σ. The rate of the new code is $\frac{kb}{n(b+l)} = \frac{k}{n}\frac{b}{b+l} = R\frac{b}{b+l}$, where R

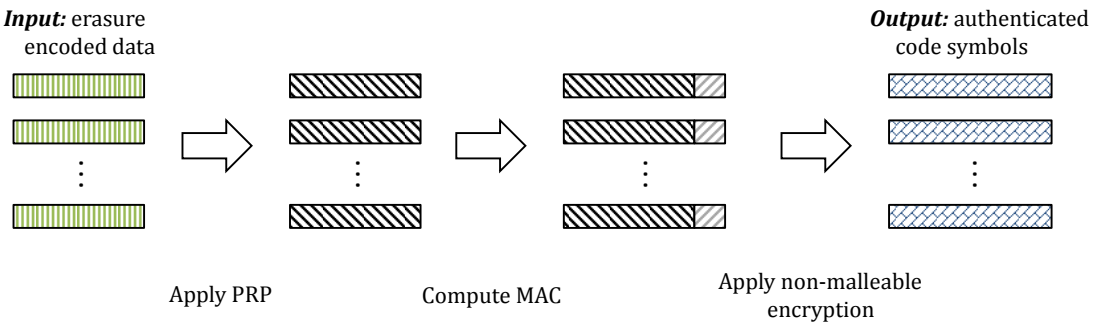


Figure 3.2: Diagram of the encoding function for AuthECC-NM.

is the rate of the original code C. Note that the decrease in the code rate is independent of the block length k and depends only on the code rate, size of Σ, the hash length λ, and the signature length s (which itself, in general, depends on λ). In practical applications, l will often be much smaller than b. For example, suppose h produces a 128-bit hash, the signature is 2048-bits (e.g., 2048-bit RSA), each code symbol is 2 megabytes in size, and we encode with a $[32, 16]$ code. In such a parameterization, the decrease in rate is $1 - \frac{(128*32+2048)/16}{2^{21}} = 1 - \frac{384}{2^{21}} \approx 0.00018$, i.e., 0.018%.

3.5.2 Non-Malleable AuthECC

One drawback of list decoding is that it is a fairly heavy-duty primitive, requiring sophisticated mathematics and algorithms. Indeed, the time to list decode a given codeword can increase dramatically as the number of errors increases (see Figure 3.3 in Section 3.7). Here we present an alternative construction that does not utilize list decoding and can be quite efficient.

Recall, that our motivation is to remove any ambiguity about the health of the data if its checksum fails to verify. To accomplish this without list decoding, we compute a signature over each symbol and perform a transformation on the signature-symbol pair such that any corruption, with high probability, corrupts both the signature and the symbol.[15] A simple way to accomplish this is to use a non-malleable cipher to encrypt the pair. Intuitively, a cipher is non-malleable if an adversary is unable manipulate a ciphertext so that a given predicate on the decrypted message evaluates to true (with an advantage non-negligibly different from simply choosing a ciphertext at random). In this case the predicate would evaluate to true if the encrypted symbol-signature pair was corrupted such that when decrypted, either the symbol is corrupted or the signature is, but not both.

[15] This is similar to the All-Or-Nothing Integrity property of entangled cloud storage, introduced in [8]. See Section 3.8 for a more detailed comparison.

Algorithm 3.3 AuthECC-NM Encode: The encoding function for AuthECC-NM.

Input: Security parameter 1^λ, secret keys k_{kdf}, k_{mac} and k_{enc}, counter ℓ, and message $m \in \Sigma^k$,
Output: Encrypted and authenticated codeword $\tilde{c} \in \tilde{\Sigma}^n$

1: Set $(c_1, \ldots, c_n) \leftarrow \mathsf{Encode}(n, k, m)$ ▷ Encode has minimum distance d
2: **for** $1 \leq i \leq n$ **do**
3: Set $k' \leftarrow kdf(k_{kdf}, \ell \circ i)$ ▷ kdf is a key-derivation function
4: Set $e_i \leftarrow f(k', c_i)$ ▷ f is a pseudorandom permutation
5: Set $\tau_i \leftarrow \mathsf{Mac}(k_{mac}, e_i \circ \ell \circ i)$
6: Set $c'_i \leftarrow e_i \circ \ell \circ i \circ \tau_i$
7: Set $\tilde{c}_i \leftarrow \mathsf{Enc}(k_{enc}, c'_i)$ ▷ Enc is a non-malleable cipher
8: **end for**
9: Output $(\tilde{c}_1, \ldots, \tilde{c}_n)$

However, this simple combination is not enough. In particular, if the non-malleable cipher is a public-key scheme then the adversary $\mathcal{A}$ can trivially independently corrupt the symbol-signature pair. In particular, if the symbol is known, then $\mathcal{A}$ can simply append a random string of the appropriate length, encrypt it, and then substitute it for the encrypted symbol in the encoding. With all but negligible probability, the random string will not be a valid signature resulting in a verification failure and the discarding of good data. Simply appending some randomness (e.g., derived from a secret shared between sender and receiver) to the symbol does not help since that randomness logically becomes part of the authentication tag and falls to the same replace-with-random-value strategy.

To overcome this, we can change how we use the counter when encoding. Specifically, we apply a key-derivation function (KDF)[16] to the counter and a secret key to derive an ephemeral key for a pseudorandom permutation (over the set of message symbols). We then apply the keyed permutation to the symbol, compute a MAC over the output and encrypt the ciphertext-MAC pair with a non-malleable cipher. Intuitively, this ensures that the adversary $\mathcal{A}$ cannot, except with negligible probability, create an input to the non-malleable cipher that will decrypt to the original data. Essentially, this construction binds together the integrity of the symbol and the signature, which contrasts with the previous construction where we completely separate the corruption of the symbols and their tags.

However, since we use a public-key non-malleable cipher, $\mathcal{A}$ can construct ciphertexts that decrypt to arbitrary messages (i.e., symbol-MAC pairs). Applying a PRP to the message before computing the MAC ensures that, when creating a ciphertext (for the non-malleable cipher), the adversary cannot control the final message that is decrypted (after inverting the PRP). If we only used a

[16] A key-derivation function is a function that takes a master secret as input along with some other (possibly public) information to derive one or more secret keys. Often, they are instantiated with a pseudorandom function.

semantically secure cipher (or some other keyed transformation), then it can be trivial to construct a ciphertext that decrypts to a specific value. In particular, suppose we have a semantically secure cipher Π (which inherently must be a randomized cipher and include randomness in the ciphertext to support deterministic decryption), we modify it to create Π' by simply fixing a message m' and a specific string of randomness r' and mapping them to a specific ciphertext c' regardless of the key. More formally, define $\mathsf{Enc}'(k, m, r)$ such that if $m = m'$ and $r = r'$ output c, else output $\mathsf{Enc}(k, m, r)$. Assuming that a well-behaved cipher always uses fresh randomness, Π' is semantically secure. But, if $\mathcal{A}$ knows the (m', r', c) triple a priori, then he can always produce a ciphertext that decrypts to m'.

This can be exploited to independently corrupt the MAC and symbol by letting the message space $M = \{m'\}$ and when given the challenge ciphertext y: taking that known ciphertext c, appending a random value (the same length as the MAC) and then encrypting the result with the non-malleable cipher. With overwhelming probability, the random value will not verify as a MAC, and the ciphertext will decrypt to the uncorrupted plaintext m'. $\mathcal{A}$ has then successfully corrupted the MAC and symbol independently. Hence, we use a PRP to make this attack infeasible.

For simplicity and consistency with the KDF, we use a MAC (a secret-key scheme) for authenticating the (encrypted) message symbol and use a secret-key non-malleable cipher for encryption.[17] The encoder is depicted in Figure 3.2 and detailed in Algorithm 3.3, and the decoder in Algorithm 3.4. Also, similar to AuthECC-LD, by computing a signature over each symbol we place an implicit lower-bound on the size of each symbol of, e.g., λ-bits to achieve λ-bit security. This minimum size also allows us to have the PRP f have the symbol alphabet Σ as its domain and range instead of needing to embed Σ in some larger input domain for f.

A practical non-malleable cipher for this purpose is to use a block cipher in CMC mode [60]. This cipher mode works by performing CBC mode encryption forward, XORing in a mask, and then performing CBC decryption in the reverse direction. The result is a secure, tweakable PRP, which necessarily implies that it is non-malleable, as shown by Halevi and Rogaway [60]. We denote our general non-malleable construction by AuthECC-NM and the instantiation with CMC-mode by AuthECC-CMC. We use AES in CMC mode in our implementation, detailed in Section 3.7.

An alternative version of this scheme is to: (1) encode the message m to get $c = (c_1, \ldots, c_n)$; (2) compute a MAC for each c_i; and (3) apply Algorithm 3.3 to the concatenated MACs (as the

[17] The combination of pseudorandom permutation and a MAC could be replaced with an authenticated block cipher mode (such as AES in Galois counter mode).

Algorithm 3.4 AuthECC-NM Decode: The decoding function for AuthECC-NM.

Input: Security parameter 1^λ, secret keys k_{kdf}, k_{mac} and sk_{enc}, counter ℓ, and codeword $\tilde{c} \in \tilde{\Sigma}^n$
Output: Message $m \in \Sigma^k$ or $\perp$

Parse $\tilde{c} = (\tilde{c}_1, \ldots, \tilde{c}_n)$ where $\tilde{c}_i \in \tilde{\Sigma}$
Set $(c_1, \ldots, c_n) \leftarrow (\perp, \ldots, \perp)$
for $1 \leq i \leq n$ **do**
    Set $c'_i \leftarrow \mathsf{Dec}(k_{enc}, \tilde{c}_i)$ ▷ Dec is a non-malleable cipher
    Parse $c'_i = e_i \circ \ell' \circ i \circ \tau_i$
    if $\mathsf{VerifyMac}(k_{mac}, e_i \circ \ell \circ i, \tau_i) = 0$ or $\ell' \neq \ell$ **then** ▷ Discard if invalid signature
        Continue at top of loop
    end if
    Set $k' \leftarrow kdf(k_{kdf}, \ell \circ i)$ ▷ kdf is a key-derivation function
    Set $c_i \leftarrow f^{-1}(k', e_i)$ ▷ f is a pseudorandom permutation
end for
Output $\mathsf{Decode}(n, k, (c_1, \ldots, c_n))$ ▷ Decode outputs $\perp$ if $< k$ of the c_i are recovered

input message). This is fundamentally the same as the construction in Algorithm 3.3, but with a level of indirection between the non-malleable cipher and the message data itself. The security analysis for this variant is similar to the analysis of the original version, which we will analyze in Section 3.6. Efficiency-wise, this scheme performs fewer encryptions with the non-malleable cipher at the cost of computing more MACs, but computing a MAC is typically faster than encrypting with a non-malleable cipher. For example, our implementation of AES in CMC mode achieves throughput of 10s of MB per second, while a secure MAC such as [92] can achieve throughput of 100s of MB per second. We do not test this scheme in our experiments.

The AuthECC-NM construction uses an authenticate-then-encrypt structure to secure the input symbol. The security of the generic composition of encrypting and authenticating (i.e., authenticate-and-encrypt, authenticate-then-encrypt, encrypt-then-authenticate) has been studied previously (see [10, 89]), concluding that the encrypt-then-authenticate construction is the most secure under generic composition. However, those analyses focused on the data *privacy* provided by the schemes while we are concerned with data *integrity*. In particular, the encryption with a non-malleable cipher is to ensure integrity against a particular form of data corruption (i.e., corruption of just the data or just tag). As shown in Figure 2 in [10], the authenticate-then-encrypt composition ensures plaintext integrity even when the MAC is just "weakly unforgeable" (which is what we use in this work).

Efficiency. To analyze the efficiency of AuthECC-NM, we note that each symbol goes through the same process of applying a PRP, computing a MAC, and then encrypting with a non-malleable cipher. Denote the time to perform these for each symbol by t_{prp}, t_{mac}, and t_{nm} respectively. When

decoding, each symbol needs: (1) to remove non-malleable encryption; (2) verify the MAC; and, finally, (3) invert the PRP, taking times t'_{nm}, t'_{mac}, and t'_{prp} (resp.). Using, again, $t_e(C)$ and $t_d(C)$ to denote the time to encode and decode for a given code C, we get the following lemma.

Lemma 3.2. *Let* AuthECC-NM *be instantiated with an* $[n, k]$ *erasure code* C *over alphabet* Σ*, where* $|\Sigma| = q$*, and utilize a PRP* f*, MAC scheme* Ψ*, and a non-malleable cipher* Π*. Then, the encoder for* AuthECC-LD *takes time* $t_e(C) + O(n(t_{prp} + t_{mac} + t_{nm}))$ *and the decoder takes time* $t_d(C) + O(n(t'_{prp} + t'_{mac} + t'_{nm}))$.

Note that often PRPs, MACs, and non-malleable ciphers (and their respective inverse/verification functions) take time *linear* in the size of their input. In this case, the inputs are of length $\log q$, since $|\Sigma| = q$. Hence, the $O(n(t_{prp} + t_{mac}))$ term in each of the running times above can be simplified to just $O(n \log q)$. However, the t_{nm} term also includes the encryption of the nonce and the MAC, giving a running time of $t_{nm} = O(\log q + \lambda)$. This gives us the following corollary.

Corollary 3.2. *Let* AuthECC-NM *be instantiated with an* $[n, k]$ *erasure code* C *over alphabet* Σ*, where* $|\Sigma| = q$*, and utilize a PRP* f*, MAC scheme* Ψ*, and a non-malleable cipher* Π*. Then, the encoder for* AuthECC-LD *takes time* $t_e(C) + O(n(\log q + \lambda))$ *and the decoder takes time* $t_d(C) + O(n(\log q + \lambda))$.

3.6 Security Analysis

Here we prove the security of our two constructions. First, we will prove that the security of AuthECC-LD reduces to the security of the signature scheme and the hash function. Second, we will prove that AuthECC-NM reduces an independently-corrupting adversary to a jointly-corrupting one, with high probability, via the non-malleability of the cipher. Finally, we will show that corruption resilience of AuthECC-NM reduces to the security of the MAC scheme. In the following, we will assume that we have a secure key derivation function (KDF) for computing the ephemeral keys from the master key and the (fresh) counter value.

3.6.1 Security of AuthECC-LD

Let $\mathsf{AuthECC\text{-}LD}^{n,k,\Sigma}_{\Pi,\mathcal{H},\mathcal{L}}$ denote AuthECC-LD instantiated with signature scheme Π, hash function family $\mathcal{H}$, using an $[n, k]$ erasure code over alphabet Σ to encode the message, and using list-decodable code $\mathcal{L}$. For the proof, we note that there are two primitives used in series that ensure security in the

scheme: the signature and the hashes. For an adversary to succeed, they must circumvent at least one of these schemes. Roughly, if the list decoding step returns incorrect authentication tags then a signature must have been forged; if the list decoding produced the correct authentication tags, then hash collisions must have been generated. We detail this next. Let t_e and t_d denote the time to encode and decode a message m using $\mathsf{AuthECC\text{-}LD}^{n,k,\Sigma}_{\Pi,\mathcal{H},\mathcal{L}}$, where Π is a signature scheme, $\mathcal{H}$ is a family of hash functions and $\mathcal{L}$ is a (ρ, L)-list-decodable code. Note that t_d includes the time to list decode the authentication information with error rate ρ and produce a list of size L. The exact values of t_e and t_d are detailed in Lemma 3.1 in Section 3.5.1.

Theorem 3.1. *Consider* $\mathsf{AuthECC\text{-}LD}^{n,k,\Sigma}_{\Pi,\mathcal{H},\mathcal{L}}$ *using an* $[n, k]$ *erasure code* C *over alphabet* Σ *to encode the message, an efficiently* (ρ, L)*-list-decodable code* $\mathcal{L}$*, a family of* (t_h, ε_h)*-collision-resistant hash functions* $\mathcal{H}$*, and a* $(t_s, q_s, \varepsilon_s)$*-unforgeable signature scheme* Π*. Then* $\mathsf{AuthECC\text{-}LD}^{n,k,\Sigma}_{\Pi,\mathcal{H},\mathcal{L}}$ *is* (t', q', ε')*-computationally-secure against error rate* ρ *where* $t' = \min\{t_h - q'(t_e + t_d), t_s - q'(t_e + t_d)\}$ *(with* $q' = q_s$*, and* $\varepsilon' = \max\{\varepsilon_h, \varepsilon_s\}$*.*

Proof. Suppose $\mathcal{A}$ is successful in $\mathsf{ChannelExp}_{\mathcal{CS},\mathcal{A}}(\lambda)$ (as defined in Definition 3.13, Section 3.4) where $\mathcal{CS}$ is $\mathsf{AuthECC\text{-}LD}^{n,k}_{\Pi,\mathcal{H},\mathcal{L}}$. Then there exists some i where the Hamming distance between c_i and c'_i is less than ρn and the receiver makes a decoding error (i.e., $m'_i \neq m_i$ where m'_i is the decoded message). Thus, at least one of the symbols in m'_i is spurious and withstood the "sieving" step in Algorithm 3.2. There are two different cases we need to consider, namely whether or not the correct authentication information (i.e., hashes and signature) was recovered or not. If the incorrect information was list decoded, then a signature must have been forged. If the correct information was list decoded, but $\mathcal{A}$ still won, then $\mathcal{A}$ must have generated a collision in the hash function h. We analyze these two cases next.

Case 1: incorrect list decoding. In this case, consider the following reduction $\mathcal{A}'$ that utilizes $\mathcal{A}$ to break Π. $\mathcal{A}'$ interacts with $\mathcal{A}$ following the steps in $\mathsf{ChannelExp}$ (i.e., performing the encoding and decoding for $\mathcal{A}$) where there are at most q' rounds, $\mathcal{A}$ runs for total time of t', and $\mathcal{A}$ wins with probability ε'. In particular, $\mathcal{A}'$ performs all of the steps of the Encode and Decode functions for AuthECC-LD but replaces the call to $\mathsf{Sign}(sk, \cdot)$ with a call to a signature oracle $\mathcal{O}_{sk}(\cdot)$ (where sk is the secret key for the signature and $\mathcal{A}'$ has the corresponding pk).

Suppose $\mathcal{A}$ wins the game and the incorrect authentication information was recovered from the list decoding. So, we have that the list decoded authentication information in round i contains a

combination of signature and hashes different from those computed over the original message m_i. Denote the original signature by σ and the new one by σ'. Let a_i denote the authentication information, sans the signature, for message m_i, and let a'_i be the corresponding authentication information decoded by $\mathsf{DecodeLD}$.

For a stateful channel, since the counter value is correct and was included in the signed data, σ' authenticates a value that was *not* queried to $\mathcal{O}_{sk}$. Moreover, we know that σ' verifies, else the decoding would have failed. Thus it must be that (a'_i, σ') is a valid message-signature forgery. Since Π is (t_s, q, ε_s)-unforgeable, it must be that $t' \leq t_s - q(t_e + t_d)$ (and $q = q'$). Moreover, since $\mathcal{A}'$ can exactly identify which (a'_i, σ'_i) pair is a forgery (by simply decoding each one), $\mathcal{A}'$ outputs a forgery whenever $\mathcal{A}$ wins; that is, we have that $\varepsilon' \leq \varepsilon_s$. Note that ε' is the advantage of $\mathcal{A}$ in succeeding in *any* round of the game $\mathsf{ChannelExp}_{\mathcal{CS},\mathcal{A}}(\lambda)$. Also note that for a memoryless channel, $q' = 1$ and we then have $t' \leq t_s - (t_e + t_d)$.

Case 2: hash collision. In this case, the list decoding step output the correct authentication information a_i but $\mathcal{A}$ still won. This implies that some spurious symbol(s) survived the "sieving" step after list decoding, but this can only occur if the spurious symbol (or symbols) when hashed gives the same value as the original symbol. That is, there must be at least one hash collision.

Hence, by simply simulating $\mathsf{ChannelExp}_{\mathcal{CS},\mathcal{A}}(\lambda)$ for $\mathcal{A}$ (for at most q' rounds), we can extract collisions for the family of hash functions $\mathcal{H}$. Namely, if a decoding error occurs in round i and we compare the original codeword c_i with the received codeword c'_i, we can determine exactly which symbols were changed and which are collisions. Thus, we have that ε', the advantage of $\mathcal{A}$, is upper-bounded by ε_h. Moreover, t' is upper-bounded by t_h minus the time to simulate the experiment. Specifically, we have that $t' \leq t_h - q'(t_e + t_d)$.

Finally, since these cases are mutually exclusive and exhaustive, we then take the maximum of ε_h and ε_s to get the advantage of $\mathcal{A}$, and we have that t' is upper-bounded by the minimum of $t_h - q'(t_e + t_d)$ and $t_s - q'(t_e + t_d)$. In each case the number of rounds in the game is upper-bounded by q. This proves the theorem. □

Corollary 3.3. *Consider* $\mathsf{AuthECC\text{-}LD}^{n,k,\Sigma}_{\Pi,\mathcal{H},\mathcal{L}}$ *using an* $[n, k]$ *erasure code* C *over alphabet* Σ*, an efficiently* (ρ, L)*-list-decodable code* $\mathcal{L}$*, a family of collision-resistant hash functions* $\mathcal{H}$*, and an existentially-unforgeable signature scheme* Π*. Then* $\mathsf{AuthECC\text{-}LD}^{n,k,\Sigma}_{\Pi,\mathcal{H},\mathcal{L}}$ *is computationally-secure for error rate* ρ*.*

3.6.2 Security of AuthECC-NM

For AuthECC-NM, we first prove that the scheme reduces an independently-corrupting adversary to a jointly-corrupting one. We accomplish this by leveraging the pseudorandom permutation, the unforgeable MAC, and the non-malleable cipher. We first, recall the definition of a non-malleable cipher in Section 3.3.3. At a high-level, a cipher is non-malleable secure (or just non-malleable) if for any PPT relation $\mathcal{R}$ and any PPT adversary $\mathcal{A}$, the adversary cannot produce a ciphertext c' where $\mathcal{R}$ evaluates to true the decryption of c' with advantage significantly greater than (essentially) choosing the ciphertext at random. Intuitively, this means that $\mathcal{A}$ cannot create/manipulate a ciphertext such that the decryption has a particular property (e.g., the last byte is zero) with significant advantage.

Referring to the detailed experiments in Definition 3.11, we use the non-malleable cipher in a more restricted context, which implies that the cipher's non-malleability is still meaningful. In particular, we have the following limitations on our adversary $\mathcal{A}$ to model our use of the non-malleable cipher in AuthECC-NM:

- The message space M output by $\mathcal{A}$ must be a subset of the set of symbols Σ associated with an instance of AuthECC-NM, rather than arbitrary messages.
- The vector output by $\mathcal{A}_2$ and $\mathcal{S}_2$ (as defined in Section 3.3.3) has length 1.
- When creating the ciphertext y for $\mathcal{A}$, we first sample a message m from M at random. Then, instead of just encrypting m with the non-malleable cipher, we apply the steps of the AuthECC-NM encoder. Specifically, we apply a PRP to m, compute a MAC over the PRP's output, append the MAC, and then encrypt with the non-malleable cipher to create the final ciphertext. This ensures that the ciphertext given to $\mathcal{A}$ is a symbol authenticated by AuthECC-NM. While this setup prevents $\mathcal{A}$ from producing proper message-MAC pairs at will (since it cannot compute the MACs), in AuthECC-NM, the MAC is never visible outside the Encode and Decode functions (which is ensured by the privacy of the non-malleable cipher) and so this restriction better models real-world interactions with AuthECC-NM.
- We limit $\mathcal{A}$ to satisfying a specific relation: the relation evaluates to true if the decrypted symbols in p and p' match but the MACs do not, or if the MACs match and the decrypted symbols do not. (This is, by a definition, the goal of an independently-corrupting adversary.) Specifically, letting $f = f(k_{prp}, \cdot)$ be a pseudorandom permutation and f^{-1} its inverse, and

letting ℓ and ℓ' be two counter values, $\mathcal{A}$ must satisfy the relation:

$$\mathcal{R}(f^{-1}, p, p') = ((f^{-1}(e) \circ \ell = f^{-1}(e') \circ \ell') \wedge (\tau \neq \tau')) \vee ((f^{-1}(e) \circ \ell \neq f^{-1}(e') \circ \ell') \wedge (\tau = \tau')),$$

where $p = (e \circ \ell, \tau)$, $p' = (e' \circ \ell', \tau')$, $e = f(k_{prp}, s)$ and $e' = f(k_{prp}, s')$ for some $s, s' \in \Sigma$ and $\tau \leftarrow \mathsf{Mac}(k_{mac}, e \circ \ell)$, $\tau' \leftarrow \mathsf{Mac}(k_{mac}, e' \circ \ell')$.

Let $\mathsf{AuthECC\text{-}NM}^{n,k,\Sigma}_{\Pi,f,\Psi}$ denote an instance of AuthECC-NM using Π as the non-malleable cipher, pseudorandom permutation f, an unforgeable MAC scheme Ψ, and encoding with an $[n, k]$ erasure code over alphabet Σ (recall that $|\Sigma| = q$ is super-polynomial in λ), and let $\mathsf{AuthECC\text{-}NM}^{n,k,\Sigma}_{\Pi,f,\Psi}(m)$ denote an encoding of m under AuthECC-NM (the security parameter is implicit in these instantiations).

Lemma 3.3. *Consider* $\mathsf{AuthECC\text{-}NM}^{n,k,\Sigma}_{\Pi,f,\Psi}$ *using an* $[n, k]$ *erasure code over alphabet* Σ *(where* $|\Sigma| = q$*), with* $\Pi = (\mathsf{Gen}_1, \mathsf{Enc}, \mathsf{Dec})$ *being a* $(t_{nm}, q_1, q_2, \varepsilon_{nm})$*-non-malleable cipher,* $f : \{0,1\}^\lambda \times \Sigma \to \Sigma$ *a* $(t_{prp}, q_{prp}, \varepsilon_{prp})$*-pseudorandom permutation, and* $\Psi = (\mathsf{Gen}_2, \mathsf{Mac}, \mathsf{VerifyMac})$ *a* $(t_m, q_m, \varepsilon_m)$*-unforgeable MAC scheme. Then, for all messages* m*, and for all PPT* $\mathcal{A}$ *independently-corrupting adversaries running in time* t*,* $\mathcal{A}$ *jointly-corrupts* $\mathsf{AuthECC\text{-}NM}^{n,k,\Sigma}_{\Pi,f,\Psi}(m)$ *with probability at least* $1 - n(1/q + \varepsilon_{prp} + \varepsilon_{mac} + \varepsilon_{nm})$ *for* $t \leq t_{nm} \leq \min\{t_{prp}, t_{mac}\}$.

Proof. Fix an adversary $\mathcal{A}$, compute $(k_{nm}, k_{mac}) \leftarrow (\mathsf{Gen}_1(1^\lambda), \mathsf{Gen}_2(1^\lambda))$, and sample $k_{prp} \leftarrow \{0,1\}^\lambda$. Let $sk = k_{nm}$ and run $\mathcal{A}^{\mathcal{O}_{sk}}(1^\lambda)$ to get $M \subseteq \Sigma$ and define $M'_\ell = \{(e \circ \ell, \tau) \mid s \in M, e = f(k_{prp}, s), \tau \leftarrow \mathsf{Mac}(k_{mac}, e \circ \ell)\}$, where ℓ is a (fresh) counter value. For pairs $p, p' \in M'$, where $p = (e \circ \ell, \tau)$ and $p' = (e' \circ \ell', \tau')$ and defining $f^{-1}(\cdot) = f^{-1}(k_{prp}, \cdot)$, define the relation

$$\mathcal{R}(f^{-1}, p, p') = ((f^{-1}(e) \circ \ell = f^{-1}(e') \circ \ell') \wedge (\tau \neq \tau')) \vee ((f^{-1}(e) \circ \ell \neq f^{-1}(e') \circ \ell') \wedge (\tau = \tau')).$$

Since the encryption scheme Π is $(t_{nm}, q_1, q_2, \varepsilon_{nm})$-non-malleable, there exists a simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$ that successfully produces a ciphertext in the experiment $\mathsf{Expt}^{\text{snm-atk}}_{\mathcal{S},\Pi,\mathcal{R}}(\lambda)$ such that $\mathcal{R}$ holds with some probability θ (where atk $\in$ {cpa, cca1, cca2}). Moreover, the probability that $\mathcal{A}$ does produces a ciphertext such that $\mathcal{R}$ holds is at most ε_{nm} greater than θ.

To calculate θ, we note that the relation $\mathcal{R}$ only holds if $\mathcal{S}$ can match the first or second elements of the pair $p = (e \circ \ell, \tau)$ (but not both). In the case of $e \circ \ell$, we note that the pair p was chosen uniformly at random from M'_ℓ; but, M'_ℓ itself may only contain a single element, so we cannot rely on its randomness. However, since e is the result of applying a PRP f with a fresh key to some code

symbol, the probability that $\mathcal{S}$ can correctly guess the value of e is at most $1/q + \varepsilon_{prp}$[18] for any $\mathcal{S}$ running in time $t \leq t_{prp}$.

For the tag τ, the probability of guessing the correct MAC for e can be no greater than the probability of forging a MAC for any message in time t, which is ε_s for $t \leq t_s$. Otherwise, we have a straightforward reduction to generate forgeries with probability greater than ε_s in time less than t_s. Thus, assuming that if $\mathcal{S}$ can match $e \circ \ell$ or τ it can make the other entry *not* match with probability 1, then success of $\mathcal{S}$ is at most $1/q + \varepsilon_{prp} + \varepsilon_s$.

For all PPT $\mathcal{A}$ running in time at most t_{nm} attacking an encoding of a message m, using $\mathsf{AuthECC\text{-}NM}^{n,k,\Sigma}_{\Pi,f,\Psi}(m)$, the probability of $\mathcal{A}$ independently corrupting a symbol is exactly equal to its ability to make $\mathcal{R}$ true for the given symbol. Taking the union over all n code symbols, we have that $\mathcal{A}$ independently corrupts at least one symbol with probability $n(1/q + \varepsilon_{prp} + \varepsilon_s + \varepsilon_{nm})$, with the restriction that $t_{nm} \leq \min\{t_s, t_{prp}\}$. □

Now that we have established that AuthECC-NM can reduce the strongest corruption patterns to the more typical ones, we will show the strength of AuthECC-NM in preventing decoding errors. Again we use the definition of $\mathsf{ChannelExp}$ detailed in Definition 3.13 in Section 3.4. Let t_e and t_d denote the times to encode and decode (resp.) an instance of $\mathsf{AuthECC\text{-}NM}^{n,k,\Sigma}_{\Pi,f,\Psi}(\lambda)$.

Theorem 3.2. *Consider* $\mathsf{AuthECC\text{-}NM}^{n,k,\Sigma}_{\Pi,f,\Psi}$ *using an* $[n,k]$ *erasure code* C *over alphabet* Σ *(where* $|\Sigma| = q$*), with* $\Pi = (\mathsf{Gen}_1, \mathsf{Enc}, \mathsf{Dec})$ *being a* $(t_{nm}, q_1, q_2, \varepsilon_{nm})$*-non-malleable cipher,* $f : \{0,1\}^\lambda \times \Sigma \to \Sigma$ *a* $(t_{prp}, q_{prp}, \varepsilon_{prp})$*-pseudorandom permutation, and* $\Psi = (\mathsf{Gen}_2, \mathsf{Mac}, \mathsf{VerifyMac})$ *a* $(t_m, q_m, \varepsilon_m)$*-unforgeable MAC scheme. Then for all PPT adversaries* $\mathcal{A}$ *running in time* t *using at most* r *rounds in* $\mathsf{ChannelExp}$*, the probability that* $\mathcal{A}$ *is* ρ*-successful against* $\mathsf{AuthECC\text{-}NM}^{n,k,\Sigma}_{\Pi,f,\Psi}$ *is at most* $\rho n r \varepsilon_m = \varepsilon'$*, with* $t \leq \frac{1}{2}(t - r(t_e + t_d)) = t'$ *and* $r \leq q_m/n = q'$*. That is,* $\mathsf{AuthECC\text{-}NM}^{n,k,\Sigma}_{\Pi,f,\Psi}$ *is* (t', q', ε')*-secure.*

Proof. Let $\mathcal{CS}$ be $\mathsf{AuthECC\text{-}NM}^{n,k,\Sigma}_{\Pi,f,\Psi}$ and suppose that we have a PPT $\mathcal{A}$ that is ρ successful in $\mathsf{ChannelExp}_{\mathcal{CS},\mathcal{A}}$ (i.e., causes a decoding error) with probability γ. We can use $\mathcal{A}$ in a simple reduction to break the security of the MAC.

In particular, we generate the appropriate k_{kdf} and k_{nm} keys and use a MAC oracle to compute a MAC over the concatenation of the encrypted symbol and the fresh counter value. Note that, for $\mathcal{A}$ to cause a decoding error, it must be that Decode accepts a spurious symbol, which can only happen if the following hold:

[18] Otherwise, we can use $\mathcal{S}$ to distinguish f from a random permutation with advantage over ε_{prp}.

1. The counter with the symbol is the correct value; and
2. The MAC (over the encryption of the symbol and the counter) is valid.

Let $e' \circ \ell' \circ \tau'$ be a spurious authenticated symbol (after removing the layer of non-malleable encryption) that is accepted by Decode. Since the counter value ℓ' is the correct value, and the counter is updated before each encoding, the symbol cannot be a replay, i.e., $e' \circ \ell'$ could not have been queried to the MAC oracle. Thus, $(e' \circ \ell', \tau')$ is a valid forgery and we can simply output it to attack the MAC.

But, we do not have access to a verification oracle and must be careful when performing the decoding of the corrupted messages output by $\mathcal{A}$. With no way to verify MACs, we are prevented from simulating Decode to determine which of r rounds of ChannelExp results in a decoding error. Simply guessing a round at random gives us a $1/r$ chance of guessing correctly but we must still simulate Decode for the other rounds.

Since $\mathcal{A}$ can win in any of the r rounds (and, indeed, may win multiple times), we guess the *first* round i where $\mathcal{A}$ wins. The subsequent rounds are irrelevant since we can obtain the needed MAC forgery in round i. Thus, to obtain a forgery, we, (1) guess the round i in which $\mathcal{A}$ will win; (2) simulate the preceding $i-1$ rounds (as detailed next); (3) in round i, guess which of the corrupted symbols in c'_i (output by $\mathcal{A}$) is a forgery (denote it σ'_f); (4) abort $\mathcal{A}$; and finally, (5) decrypt σ'_f to get $e'_f \circ \ell'_f \circ \tau'_f$, output $(e'_f \circ \ell'_f, \tau'_f)$ as our MAC forgery and exit.

To simulate the first $i-1$ rounds, in round j, we simply save the authenticated codeword c_j and compare it symbol-by-symbol with the corrupted codeword c'_j output by $\mathcal{A}$. Any symbols in c'_j that do not have a match in c_j (and any symbols that are duplicates) are discarded. The remaining symbols are then decoded and the result is given to $\mathcal{A}$. Since $\mathcal{A}$ does not win in round $j < i$, the distribution of inputs is identical to the expected distribution (i.e., since $\mathcal{A}$ does not win in around $j < i$, any corrupted symbols are *not* successful MAC forgeries and would be discarded).

If we guess correctly (which happens with probability $1/r$), then $\mathcal{A}$ wins with probability γ and we produce a MAC forgery with probability $\gamma/(\rho n r)$ (since we can just guess at random which of the, at most, ρn corrupted symbols are a successful forgery). If we guess a round i' (where $\mathcal{A}$ wins) that is too small (i.e., less than the winning round i), then we succeed with probability 0 since we know that $\mathcal{A}$ produces no successful forgeries in that round. If our guess is too high, then we will inadvertently *reject* a corrupted codeword that causes a decoding error and thereby miss a successful MAC forgery. This, however, deviates from the distribution of inputs expected by $\mathcal{A}$ and may even be detectable by $\mathcal{A}$. In this situation, we can have no expectations about the behavior of $\mathcal{A}$.

To remedy this, we have the reduction proceed as before but we also track how many steps $\mathcal{A}$ has executed. If $\mathcal{A}$ exceeds the maximum number of steps t, then we terminate $\mathcal{A}$ and abort, ensuring that our reduction takes at most $2t + r(t_e + t_d)$ steps.[19] Assuming that if we select an i that is too large, then $\mathcal{A}$ never wins, we have that our success probability is $\gamma/(\rho nr) \leq \varepsilon_m$, which gives $\gamma \leq \rho nr\varepsilon_m$. If $\mathcal{A}$ runs in total time of t, then our reduction runs in time at most $2t + r(t_e + t_d)$ where t_e and t_d are the time to encode and decode. Hence, $t \leq \frac{1}{2}(t_m - r(t_e + t_d))$, and we have $rn \leq q_m$ since there are r rounds with at most n MAC oracle queries each. □

Corollary 3.4. *Consider* $\mathsf{AuthECC\text{-}NM}^{n,k,\Sigma}_{\Pi,f,\Psi}$ *using an* $[n, k]$ *erasure code* C *over alphabet* Σ *(where* $|\Sigma| = q$*), with* $\Pi = (\mathsf{Gen}_1, \mathsf{Enc}, \mathsf{Dec})$ *being a non-malleable cipher,* $f : \{0,1\}^\lambda \times \Sigma \to \Sigma$ *a pseudorandom permutation, and* $\Psi = (\mathsf{Gen}_2, \mathsf{Mac}, \mathsf{VerifyMac})$ *an existentially-unforgeable MAC scheme. Then* $\mathsf{AuthECC\text{-}NM}^{n,k,\Sigma}_{\Pi,f,\Psi}$ *is computationally secure.*

Proof. The same reduction above does not work here as we do not have an a prior bound t on the running time of $\mathcal{A}$. But, we can simply observe that for a decoding error to occur, a spurious symbol (with the proper counter value) must be accepted by the decoder: i.e., it must have a MAC that verifies. Since the symbol is spurious (i.e., not created by Encode), it must have a forged MAC, which can only happen with negligible probability. □

Finally, we note that if we replace the MAC scheme with a $(t_s, q_s, \varepsilon_s)$-unforgeable signature scheme, we get a much tighter reduction.

Theorem 3.3. *Consider* $\mathsf{AuthECC\text{-}NM}^{n,k,\Sigma}_{\Pi,f,\Psi}$ *using an* $[n, k]$ *erasure code* C *over alphabet* Σ *(where* $|\Sigma| = q$*), with* $\Pi = (\mathsf{Gen}_1, \mathsf{Enc}, \mathsf{Dec})$ *being a* $(t_{nm}, q_1, q_2, \varepsilon_{nm})$*-non-malleable cipher,* $f : \{0,1\}^\lambda \times \Sigma \to \Sigma$ *a* $(t_{prp}, q_{prp}, \varepsilon_{prp})$*-pseudorandom permutation, and* $\Psi = (\mathsf{Gen}_2, \mathsf{Sign}, \mathsf{Verify})$ *a* $(t_s, q_s, \varepsilon_s)$*-unforgeable signature scheme. Then for all PPT adversaries* $\mathcal{A}$ *running in time* t *using at most* r *rounds in* $\mathsf{ChannelExp}$*, the probability that* $\mathcal{A}$ *is* ρ*-successful against* $\mathsf{AuthECC\text{-}NM}^{n,k,\Sigma}_{\Pi,f,\Psi}$ *is at most* $\varepsilon_s = \varepsilon'$*, with* $t \leq t_s - r(t_e + t_d) = t'$ *and* $r \leq q_s/n = q'$*. That is,* $\mathsf{AuthECC\text{-}NM}^{n,k,\Sigma}_{\Pi,f,\Psi}$ *is* (t', q', ε')*-secure.*

Proof. The reduction proceeds similarly to the proof of Theorem 3.2 but, since we use a signature scheme, we can verify the signature on each authenticated symbol and detect and successful/unsuccessful forgeries. That is, we can simulate Decode exactly: at each step, the public key of the signature scheme is used to detect and discard corrupted symbols. Additionally, as before, we track

[19] The factor of 2 comes from incrementing the counter at each step of $\mathcal{A}$.

each c_i given to $\mathcal{A}$ as well as the corresponding c'_i and compare them to find which symbols have changed. Then, checking the signature on only the changed symbols (that have the correct counter value) will reveal which symbol-counter-signature triple is a valid signature forgery (if any). Thus we have that we succeed in forging a signature exactly when $\mathcal{A}$ wins, and if $\mathcal{A}$ runs in time t then the reduction runs in time $t - r(t_e + t_d)$ with $r \leq q_s/n$, giving a success probability of $\varepsilon' = \varepsilon_s$. □

3.7 Experiments

To show the practicality of our constructions, we implemented both AuthECC-LD and AuthECC-NM, with the latter instantiated with AES in CMC mode, and show the results of several experiments here. We denote these schemes as LD and CMC, respectively. The experiments were performed on an Intel Core i5 processor at 2.6GHz with 4GB of RAM and running Arch Linux with the 3.14.19 kernel. The code was written in C and compiled with `gcc` version 4.9.1 with the '-O2' optimization flag. We used the Guruswami-Sudan list decoding algorithm (for Reed-Solomon codes) as our list-decodable code, as implemented in the decoding library by Guillaume Quintin [137]. The implementation includes Kötter's polynomial interpolation algorithm (described in [83], detailed in [109]), and a variant of the Roth-Ruckenstein root-finding algorithm by Berthomieu et al. [16]. Finite field arithmetic is performed in $GF(2^{128})$ using the MPFQ finite field library [163]. Where applicable, we encode messages using a systematic Reed-Solomon code using the Jerasure library [71].

We used SHA256 for the collision-resistant hash function, UMAC for the message authentication code [92], OpenSSL version 1.0.1i for AES encryption when implementing CMC mode, and PBKDF2 for key derivation [74]. We used 2048-bit RSA keys for digital signatures, again using the OpenSSL implementation, and AES in CBC mode with a random IV for our PRP. We also implemented the scheme by Micali et al., where the message is signed before applying a list-decodable code (as described in [111]) and refer to it as STE for "sign-then-encode."

We do not benchmark the Guruswami-Rudra list decoding algorithm [58] for folded Reed-Solomon codes (which can correct up to a $(1 - R)$-fraction of errors, the theoretical limit for any ECC) due to lack of a ready implementation of the scheme. We also do not benchmark decoding using multiple polynomials (used in [34]) via the Cohn-Heninger algorithm [31] to keep our implementation simple.

In Figure 3.3 we see the time to decode for LD, CMC, and STE. The horizontal axis of the graph gives the *error rate*: the fraction of code symbols (and encoded tags) corrupted. For CMC, the

Figure 3.3: Graph of time to decode with a variable error rate. For a $[32, 16]$ code using the Guruswami-Sudan list decoding algorithm, the maximum number of errors that can be corrected is $32(1-\sqrt{0.5}) \approx 9$. Input file was 1KB in size.

corruption was applied directly to the symbols; for LD, the corruption was applied to the symbols and the *same fraction* of errors was applied to the encoded authentication tags. The graph clearly shows an almost constant decoding time for CMC as the number of errors increase, and a much larger increase for both LD and STE. In particular, when correcting errors close to and beyond the unique decoding radius (i.e., when the error rate is near or greater than 0.25), the running time increases dramatically. Also note that LD and STE perform almost identically with LD having a small speed advantage (not visible in the graph).

Since the time to list decode is heavily dependent on the code parameters n and k, we chose the input message size and $[n, k]$ values to ensure that LD and STE have the same $[n, k]$ when using a list-decodable code. (Both LD and STE operated over $GF(2^{128})$.) Suppose the input size is l, the counter c is 16 bytes, and the code rate is $R = 0.5$, then, STE must use parameters $k' = (l + 16 + 256)/16 = l/16 + 17$ and $n' = k'/R = l/8 + 34$ to encode the message. For LD, if the message itself is encoded with an $[n, k]$ code, then the authentication tags together are $32n + 16 + 256$ bytes long (the first term for the SHA256 hashes and the latter two for the counter and the signature). Since each element in $GF(2^{128})$ is 16 bytes in size, LD must have $k'' = (1/16)(32n + 272) = 2n + 17$ and $n'' = k''/R = 4n + 34$. To ensure that $k'' = k'$, we chose to have an input file of 1KB and parameters $n = 32$ and $k = 16$. This in turn gives $k'' = k' = 81$ and $n'' = n' = 162$. As a result, LD and STE schemes have virtually identical performance.

The sharp increase is not intrinsic to the list decoding per-se, rather it is due to the particular choice of subroutines, in particular, the use of Kötter's algorithm for interpolation [83]. Kötter's

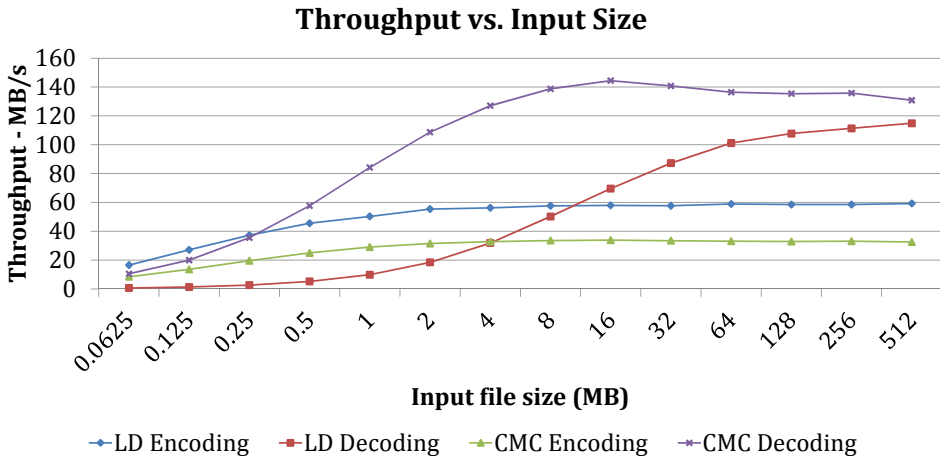


Figure 3.4: Graph of speed of list decoding and CMC with variable input sizes and $[32, 16]$ code. Decoding speeds relative to 2 errors in the codeword.

algorithm performs bivariate polynomial interpolation and takes as a parameter the multiplicity m of the roots of the polynomial (which is chosen carefully in Reed-Solomon decoding). The algorithm runs in time $O(n^2m^4)$, where n is the codeword size. The multiplicity is calculated $m = 1 + \left\lfloor \frac{kn-(n-e)}{(n-e)^2-kn} \right\rfloor$, where e is the number of errors in the received codeword. As the number of errors approaches the limit of $n(1-\sqrt{R})$ (where R is the code rate), since $kn = Rn^2$ and $(n-e)^2 = (n-n+n\sqrt{R})^2 = Rn^2$ the multiplicity increases without bound (since the denominator goes to 0). Note that $e = n(1-\sqrt{R})$ is the number of errors corrected *in the limit*. The actual number corrected will be $\lfloor e \rfloor < n(1-\sqrt{R})$. A different interpolation algorithm, such as [84], would provide different asymptotic performance.

In Figure 3.4, we see that both LD and CMC achieve practical throughput in both encoding and decoding. Of note is the fact that, for both, the throughput of encoding and decoding levels out for large files. This hints at the fact that, for fixed $[n, k]$, the use of hashing combined with a list-decodable code in LD results in an overhead *linear* in the file size. In particular, the hashing takes a linear amount of time and the encoding/decoding of the list-decodable code takes *constant* time since the input to the code is of a fixed size. The STE approach, on the other hand, to accommodate large files would require either increasing n and k when encoding the file—leading to a quadratic increase in the encoding and decoding times—or (keeping n and k fixed) the size of the underlying field would need to increase. Indeed, the field elements would need to be very large to accommodate these big slices of data. Hence, we omit STE from this comparison.

In $GF(2^m)$, multiplication can, asymptotically be done in $O(m \log m \log\log m)$ time by regarding $a \in GF(2^m)$ as a polynomial in $GF(2)[x]$ and using the Schönhage-Strassen multiplication algorithm.

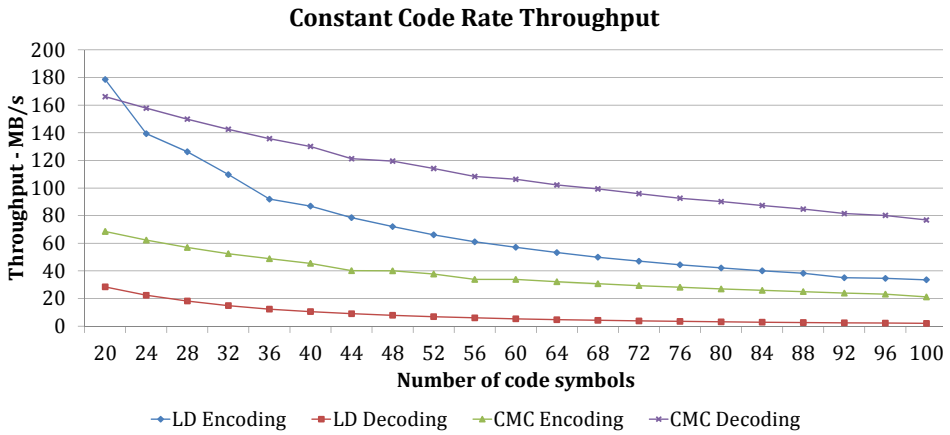


Figure 3.5: Graph of speed of CMC and list decoding with a constant rate and variable number of symbols. Input file was 2MB, the code rate was 0.75, and there were no errors.

Addition and subtraction takes $O(m)$ time by simply adding the two elements as regular integers and then subtracting the modulus 2^m (again, as a regular integer) if the result is bigger than 2^m. Hence, STE's throughput *decreases* as the size of the file (and, hence, the size of field elements) increases, which is shown in Table 3.4. Note that to maintain the same n and k for each field, we must increase the size of the input file. This has the side effect of increasing the time to compute and verify a signature, since the file is hashed before being signed. However, in this case, due to the small input sizes, the increasing file size has a negligible impact on signing and verification efficiency. For example, in the same set of experiments, the largest file on average took 2.0ms to sign while the smallest file took 2.3ms. Our AuthECC-LD construction avoids this scalability problem and, indeed, can have the input file grow quite large (with fixed n and k) without increasing the cost of list decoding.

Figure 3.5 shows the performance of LD and CMC when encoding with a fixed rate while increasing the number of message and code symbols. The throughput of LD when encoding and decoding decreases quadratically with n, which is expected. Indeed, decoding throughput goes to

Table 3.4: Encoding and decoding times of STE for a $[288, 144]$ code with no errors in the received code word over fields of increasing size. Times are the average of 50 trials. Input sizes *do not* include the signature but *do* include the counter.

Field	Encoding time	Decoding time	Input size
$GF(2^{16})$	2.7 ms	12.1 ms	32 bytes
$GF(2^{32})$	2.6 ms	13.9 ms	320 bytes
$GF(2^{64})$	3.2 ms	25.7 ms	896 bytes
$GF(2^{128})$	3.7 ms	41.0 ms	2048 bytes

near 0 as the number of symbols increases. In contrast, even though it slows down as k and n increase, CMC mode achieves reasonable encoding speeds across all inputs. Moreover, the decoder keeps comparatively high throughput, even for large files. One reason for this gap in performance between CMC encoding and decoding is the fact that both the PRP and CMC mode encryption use CBC encryption, and CBC mode can be easily parallelized when *decrypting* but not when encrypting.

3.8 Previous Work

There have been many instances of systems that combine erasure/error correcting codes and cryptography together. We overview some of them here.

3.8.1 Computationally-bounded Channels

In this work, we prove our constructions secure against a particular computationally-bounded adversary. This is a more limited adversary than the general, information-theoretic approach (implicitly) taken in information theory. But, since attacks in the real world are restricted to feasible computations, this limitation gives a more accurate model of the world. This approach was first proposed in [98], in which Lipton introduces and formalizes the concept of a computationally-bounded adversarial channel (this was further developed in the unpublished manuscript [55]). In this, he shows via a simple construction (called "code scrambling"), that any code that has success probability q over the binary symmetric channel with error probability p, has success probably q over *any* (computationally-bounded channel) with error probability p. The construction works by simply generating and applying a random permutation of n elements to the code word than then adding in a random pad. This requires $\Theta(n \log n)$ random bits, but by assuming the existence of a PRG with security parameter λ, then the construction only needs $O(\lambda)$ truly random bits (where $\lambda \ll n$). He also presents a construction that divides the input into blocks of size $a \log n$ (for a suitably chosen a), independently encodes them, and then scrambles them all together. Before the scrambling, the construction could handle $O(\log n)$ errors in the worst case, but after scrambling it can handle $O(n)$ errors. Lipton in this work reduces adversarial corruption to random corruption, in the computational setting, but does not enhance the error-correction in general (i.e., cannot correct more errors) as we do in this work.

Later, Langberg in [95] defines the notion of a private code: i.e., a code that takes a secret key as a parameter. In this work, he proves that any code over the binary symmetric channel (with

error probability p) can be turned into a private code over an adversarial channel that corrupts at most pn symbols. This is done, as in [98], via random permutation over the code symbols. The difference between Langberg's work [95] and Lipton's [98], is that the former makes no cryptographic assumptions (so the key must be $\Theta(n \log n)$ bits). In addition to this simple construction, Langberg proves a lower bound of $\Omega(\log n)$ on the number of random bits needed to be secure against an adversarial channel. Additionally, he provides a construction that meets this lower-bound (i.e., requires only $O(\log n)$ random bits). This construction operates via a careful partitioning the space of code words for a list-decodable code. Decoding is performed with a maximum-likelihood decoder. As in [98], this reduces adversarial corruption to random corruption but does not allow the code to correct additional errors.

In [111], Micali et al. describe a game for a stateful, computationally-bounded adversarial channel and provide a construction that is secure against that channel. They describe a game—modeled after cryptographic games—involving multiple rounds where the channel chooses a message to be encoded, corrupts the encoding, and then gives the result to the receiver. The adversary wins if at any point the decoder decodes to an incorrect message. We use this model as the basis for our model in Section 3.4. Micali et al. also provide a construction that is secure against their adversary: essentially, the scheme is a combination of digital signatures and list decoding. Specifically, they sign a message before encoding it with an efficiently list-decodable code. When decoding, the received codeword is list decoded and the signature is used to disambiguate the list and achieve unique decoding. This is basic the same construction found in the earlier paper Lysyanskaya et al. [106] that applied the same technique to authenticate a group of packets. In both works, adversarial success reduces to forging a signatures and the use of list decoding allows for the correction of more errors. This is the "sign-then-encode" approach and is used as part of the AuthECC-LD construction.

An *adversarial error correcting code* (AECC) is defined by Bowers et al. in [21] to be a code that can tolerate some amount of adversarial corruption. They say that a particular AECC is (β, δ)-bounded if, for an $[n, k]$ code, and a computationally-bounded adversary $\mathcal{A}$, $P[(c, c') \leftarrow \mathcal{A}(1^\lambda); \Delta(c, c') \leq \beta n] - \delta$ is negligible in λ. That is, the probability that the $\mathcal{A}$ causes a two codewords c and c' to decode to different messages, where the distance between c and c' is less than βn, is negligibly different from some probability δ. This models adversarial behavior against proofs-of-retrievability (PoRs, described later). A PoR provides two guarantees:1. if the corruption is below some threshold t, it can be repaired; and 2. if the corruption is above the threshold t, it will be detected with high probability.

This motivates $\mathcal{A}$ to minimize $\Delta(c, c')$ while the code designer wants to maximize β. This definition, however, allows for the possibility that if $\Delta(c, c') > \beta n$ and $\Delta(c, c') \leq n - k$, then $\mathcal{A}$ can easily cause the codewords to decode to different messages. That is, the definition allows for a *gap* in the error correction capacity of the underlying code and the tolerance to adversarial corruption. In a PoR, this does not matter since the corruption of more than βn symbols would (presumably) be easily detected, but we want to preserve the integrity of encoded messages *regardless* of the amount of corruption (up to the error correcting capacity of the code).

3.8.2 General Combination of Codes and Cryptography

Randomization. In work combining cryptography and error correcting codes, Luby and Mitzenmacher in [104] describe a "verification decoding" algorithm based on belief propagation for LDPC codes. They assume that a corrupted packet takes on a uniformly random value (rather than an adversarially chosen value). The decoder performs two steps repeatedly: (1) if the sum of all neighbors of a parity node is 0, then mark all nodes as verified, (2) if a parity node has a single unverified neighbor, then it is marked as verified and its value is set to the sum of all other neighbors. With this decoder, they are able to correct errors over the binary symmetric channel when previously, only erasures could be corrected. Building on this, they independently rediscover the result of [98] and reduce an adversarial channel to the binary symmetric channel via a random linear transformation applied to all code symbols followed by a random permutation. (The same authors also use this technique in [114] to reduce adversarial corruptions to random ones for a code based on invertible Bloom lookup tables; see [114] for details.)

In [150], Smith seeks to protect against adversarial errors while using few random bits and avoiding any unproven computational assumptions (e.g., the existence of a cryptographically strong PRG). In particular, he draws a random permutation from a t-wise independent family, where $t = o(n)$.[20] Moreover, this is accomplished using $n + o(n)$ random bits instead of the previous best of $\Theta(n \log n)$. The basic construction is the same as that of Lipton in [98], where a plain ECC is applied to a message and then a random permutation is applied to the codeword. The resulting codes are capacity approaching and correct pn errors with probability exponentially close to 1. Our work makes additional assumptions (e.g., the existence of secure MACs) allowing us to use many fewer bits, i.e., just $\lambda \ll n$ bits.

[20] Such a permutation can be generated using $O(t \log n)$ bits using a specific (non-cryptographic) PRG.

Guruswami in [57] presents list decoding in the presence of *side information* where the sender and receiver have a noisy primary communication channel and a non-noisy (and likely expensive) side channel for transmitting additional information. Our schemes can be modeled in this context with the keys being transmitted on a (noiseless) side channel immune to adversarial eavesdropping. Guruswami proves that for deterministic schemes, the amount of side information required to allow unambiguous decoding is nearly equal to the size k of the message itself. For a randomized scheme (with a small probability of decoding failure, but not decoding error) the amount of side information required is $\Omega(\log k)$ and he provides a scheme that meets this bound. Guruswami's schemes operate in the presence of a computationally-*unbounded* adversary, so his results are not directly applicable here. Indeed, our schemes have key lengths that are independent of the message length k (but logarithmic is the alphabet size), provided that k is polynomially bounded.

Jaggi et al. in [70] provide information theoretically secure network codes that can tolerate byzantine nodes in the network. If the network has capacity c and the adversary can eavesdrop on all links and jam a z-fraction of the links, then their codes achieve a rate of $c - 2z$. If the adversary has limited knowledge, e.g., can only eavesdrop on a fraction of the links, then their codes have rate $c - z$. The also provide construction can also utilize a secret, noiseless side channel between the sender and receiver to facilitate decoding. In particular, the receiver uses list decoding to recover a list containing the correct decoding and then uses a hash of the original message (sent over the side channel) to disambiguate the list decoding.

Cryptography in Codes. There are several examples of schemes that combine cryptography and ECCs together. An example where cryptography is used in the heart of an ECC is found in [127] (further developed in [128]) where Perry et al. present a new rateless error correcting code called a *spinal code*. Spinal codes work by breaking the input message of n bits into blocks of k bits and then applying a random hash function to each block to produce a real number in the interval $[0, 1)$ (called a *spine*). The encoder then makes passes over the spines and applies a deterministic function to map a portion of each spine to an output bit.

Guruswami and Smith present several results in [59] for codes against computationally-bounded adversaries that corrupt up to a p fraction of the code symbols. First, they provide constructions for codes that approach channel capacity for additive (i.e., oblivious) channels. Second, they give polynomial time list-decodable codes with optimal rate for log-space adversaries. Their construction

for list-decodable codes can be extended to handle an adversary with n^c space for any fixed $c > 1$ (but c must be fixed beforehand). For the first two constructions, they only assume the existence of one-way functions, and for the latter they assume the existence of PRGs that can fool circuits of size n^c. In their constructions, the sender and receiver do *not* have pre-shared secret unknown to the adversary. They hide the secret randomness in the message itself such that the adversary (with only log-space) cannot locate it, but the receiver (who has super-logarithmic space) can.

Computational Locally Decodable Codes. In [26], Chandran et al. define locally-updatable, locally-decodable (error correcting) codes (LULDCs). Their constructions have constant code rate and allow for local updates to an encoded message in addition to having local decodability (i.e., any message bit can be recovered by querying a few code bits, the number of bits queried is the *locality*). Write operations in their schemes have a locality of $O(\lambda \log k)$ and a read operations have a locality of $O(\lambda \log^2 k)$. They accomplish this against computationally-unbounded adversaries. However, the adversaries are limited so that they do not corrupt "too many" of the recently updated bits: that is, only a few of new bits can be corrupted, but many of old bits can be corrupted. This is accomplished by having a hierarchy of levels (with a logarithmic number of levels) where each level twice the size of the one below it. Updates start in the lowest (and smallest) levels and percolate up to the higher levels as more updates are made with each level encoded with a locally-decodable code. Against computationally-bounded adversaries, they can detected arbitrary errors (via MACs) and can correct a limited class of corruptions.

Ostrovsky et al. present constructions in [119] for private locally decodable codes against computationally-bounded adversaries. By assuming the existence of one-way functions, they construct asymptotically good locally decodable codes over a binary alphabet. They can correctly decode any bit after querying $\omega(\log^2 \lambda)$ bits in a codeword, with probability greater than $1 - \lambda^{-\omega(1)}$. If the sender and receiver have a shared state (e.g., a public counter), then their query complexity is $\omega(\log \lambda)$. Furthermore, they show that $\omega(\log \lambda)$ bits are necessary to achieve a negligibly small probability of decoding error.

3.8.3 Secure Storage

Cloud Storage. There have been many works combining cryptography and erasure/error correcting codes to construct reliable, secure storage (especially cloud storage). One example is [17], where

Bessani et al. utilize a combination of a "cloud-of-clouds" and byzantine agreement protocols to overcome the (possibly byzantine) failure of an individual cloud provider. Using byzantine agreement protocols, they are limited to at most 1/3 of the servers being malicious, but can handle arbitrary responses from the malicious servers. Data is encrypted for privacy and then the key is distributed among the servers using Shamir's secret sharing scheme [146]. The data is then encoded with an erasure coding for redundancy and each symbol is hashed and signed to detect corruption.

In [88], Krawczyk uses an information dispersal algorithm (e.g., encoding the file with a Reed-Solomon code) combined with a perfect secret sharing scheme to securely and reliably distribute a (large) secret. He first chooses a random key K for encrypting the secret message m. The ciphertext c is then encoded using a (t, n)-information dispersal algorithm, where any t out of n shares can be used to reconstruct the message. The key is then encoded with a (t, n)-perfect secret sharing scheme and each share of the encrypted message is paired with a share of the key and the pairs are then distributed among n servers. In the same work, Krawczyk presents a *robust* secret sharing scheme, where he augments his previous scheme by computing the hashes of some erasure encoded symbols and then using an error-correcting code to encode each hash. The shares of each encoded hash are then distributed with the symbols. This technique is an application of distributed fingerprints [87] (also by Krawczyk) and provides security against corrupting adversaries.

In [25], Cao et al. utilize LT codes, bilinear maps, and homomorphic MACs to produce a secure cloud storage scheme that allows for asymptotically efficient encoding and decoding (via the LT codes), and allow for the data to be repair if any servers are lost (there is a special server that handles the repair operations). In this setting, they consider computationally-bounded adversaries, but only consider attacks against the data servers and not against the repair server.

Goodson et al. give several protocols in [51] that ensure consistency and integrity of data stored in a collection servers in the presence of (both) byzantine clients and servers. The protocols allow for "linearizable and wait-free read-write objects." For reliability, files are erasure coded and each code symbol is hashed and the concatenation of the hashes is replicated (called a *cross-checksum*). Messages between clients and servers are authenticated via pair-wise sharing of secrets among clients and servers.

Cachin and Tessaro combine erasure codes and Merkle trees (with code symbols as the leaves of the Merkle tree) in [24] to authenticate data written by a client to a group of servers. The servers use a reliable broadcast to transmit their symbol and Merkle tree path to the other servers so that each

server can verify the authenticity and integrity of its symbol. They also use threshold cryptography to encrypt the file and distribute the key to the servers such that at least 1/3 of the servers are needed to reconstruct the key. In a follow-up to [24], Hendricks et al. in [63] utilize homomorphic "fingerprinted cross-checksums" to drastically reduce the inter-server communication in the protocols. The checksums are keyed, homomorphic universal hash functions and allow each server to efficiently verify that their code symbol is correct. Further follow-up includes [62] by the same authors.

The work of [156] by Storer et al. gives a secure archival storage system with a "write-once-read-maybe" data model with data lifetimes on the order of decades. A file is split into pieces (called *fragments*) using an information theoretically secure secret sharing scheme for privacy. The fragments are then split again using a threshold secret sharing scheme (e.g., Shamir's) for availability; these pieces are called *shards* and are distributed among "archives" (each "archive" is a data repository and separate security domain). Distributed RAID techniques are used for fault-tolerance among the archives.

Other efforts to secure cloud storage include the work of Wong et al. in [164] where a file is encoded by dividing it into blocks and using a systematic $[n, k]$ Reed-Solomon code on each block over $GF(2^m)$ with $m = 8$ or 16. The encoding matrix is kept secret. They also integrate a proof of data possession (PDP) in their scheme where challenge tokens are created by taking a random subset of blocks, interpreting them as coefficients of a polynomial and then evaluating the polynomial at random points. See the original paper [164] for more details.

Proofs-of-Retrievability. A proof-of-retrievability (PoR), first defined and explored by Juels and Kaliski in [72], is scheme that makes heavy use of cryptography and error correcting codes to secure remote storage. PoRs consist of encoding, decoding, and audit protocols, the last of which is used to detect data corruption. Intuitively, a PoR guarantees that if the adversary does not corrupt too much, then the damage can be repaired; conversely, if the damage cannot be repaired, then it will be detected with high probability.

There have been several follow-up works to [72]. Shacham and Waters in [145] present two PoRs: the first is secure in the random oracle model and has short audit query and responses, the second is secure in the standard model and has short audit responses but longer queries. The work of Bowers et al. in [20] combines Reed-Solomon codes and universal hashing to produce secure, homomorphic MACs over the data that give an efficient audit protocol. In addition, the adversary considered in [20]

is a *mobile adversary* that may (over time) corrupt all servers but at any given time it only has up to a constant fraction compromised. PoRs are further explored in [21] by Bowers et al. where they provide rigorous theoretical framework for designing PoRs and provide improved variants of [72] and [145]. In [142], Sarkar and Safavi-Naini present a PoR that utilizes Raptor codes. Specifically, they apply an erasure code to the data and apply a homomorphic authenticator to each symbol and the audit protocol then performs LT encoding over the erasure encoded data. This allows an unbounded number of challenge messages to be created as well as efficient encoding and decoding.

Much early work for PoRs assumed that the encoded data was static, and the scheme could not be readily extended to the dynamic case. In [147], Shi et al. use a hierarchical log structure, where each level is erasure encoded, combined with Merkle trees and MACs to achieve an efficient dynamic proof-of-retrievability. They achieve efficiency logarithmic in the number of blocks (multiplied by the security parameter) for both bandwidth and server computation with audit costs also logarithmic in the number of blocks and quadratic in the security parameter.

Entangled Storage. In a follow-on to PoRs, Ateniese et al. define *entangled cloud storage* in [8], where a client (or a group of clients) combine their files together (i.e., "entangle" them) such that it is infeasible to corrupt or destroy a large part of the entangled data without affecting *all* of the input files (called "all-or-nothing integrity"). The entanglement procedure is distributed among the clients and no coordination is needed among the clients for any of them to recover their own files from the entangled storage. The goals of the entangled storage are similar to one of our goals in designing AuthECC-NM in that we strive to force the adversary to corrupt both the code symbol and its MAC (and the counter). They achieve their construction through a combination of pseudorandom pads and polynomial interpolation (where the individual files are the result of evaluating the polynomial at a specific point). Our construction of AuthECC-NM, uses a non-malleable cipher to provide entanglement since, intuitively, the non-malleable cipher cannot be manipulated or mauled to corrupt only the symbol or only the MAC.

Non-malleable Codes. Looking at cryptography and codes, there are primitives known as *non-malleable codes*, first defined and constructed Dziembowski et al. in [39]. The work was further developed by Faust et al., described in [40], Cheraghchi and Guruswami in [28]. Non-malleable codes seek to encode a message such that, if it is tampered with, it will decode to a message *unrelated* to

the original message. They were designed to protect against malicious hardware tampering so that any manipulation by an adversary results in a random internal state (rather than an adversarially chosen one). Intuitively, non-malleable codes seek to *exacerbate* errors rather than correct them.

3.8.4 Secure Networking

Multicast Authentication. Lysyanskaya et al. in [106], couple digital signatures with list decoding to achieve error correction beyond the unique decoding radius (i.e., half the minimum distance of the code). Specifically, they sign the input message and then apply an efficiently list-decodable code (e.g., a Reed-Solomon code) to the message-signature pair. When decoding, the codeword is list decoded and the correct entry in the list is located by checking the signatures. This technique of using signatures to achieve unique decoding from list decoding over a computationally-bounded channel was independently discovered by Micali et al. in [111]. This latter work also developed a theoretical foundation for a computationally-bounded channel that attacks a block code.

In [121], Pannetrat and Molvathe combine error correcting codes with cryptographic hashes and digital signatures for efficient multicast authentication against adversarial *erasure* channels (the channel can erase up to a constant fraction p of the packets). Specifically, they break the packets up into blocks of n packets, hash the packets, sign the hashes, and then use a systematic erasure code to encode the hashes and signature. The parity symbols generated from the systematic encoding are then concatenated and divided up, evenly, among the packets. This results in just tens of bytes overhead per packet. Given $(1-p)n$ packets, the hashes are computed for the packets and the parity symbols for the encoded hashes and signature are reconstructed. Since the erasure coding was systematic, the combination of $(1-p)n$ hashes with the rebuilt parity symbols is enough to recover all of the hashes and the signature. The signature is then verified and the packets are all thrown out if the signature is invalid.

Distillation codes are a class of fixed-rate error correcting codes, first described by Karlof et al. in [76], that can withstand adversarial corruptions of the symbols including injection of extra symbols into the codeword. Essentially, distillation codes are an erasure code augmented with a one-way accumulator and a signature scheme. They use a Merkle tree as an example accumulator. The input message is "tagged" (e.g., signed) and then erasure encoded; the encoded symbols are then used as the leaves in a Merkle tree. Each symbol is sent with the neighbors on the path from the symbol to the root of the tree. When decoding, the received symbols are partitioned into groups based on the

root hash value computed when verifying each symbol. Each group is then erasure decoded, and for any successful decodings the tag is verified and all groups with an invalid tag are discarded. The decoder then randomly selects one of the remaining groups and outputs the decoding.[21] This work is designed to allow for arbitrary errors in addition to *pollution* attacks, where the adversary injects extra (spurious) symbols into the stream. Their scheme is secure over a computationally-bounded channel, though this is not stated nor formalized.

The work in [91] by Krohn et al. provides an efficient construction for online verification of erasure encoded symbols produces by a rateless code, such as Raptor or Online codes, for content distribution in peer-to-peer networks. Their adversary is one that sends spurious blocks to clients requesting a given file. The authors use homomorphic hashing—via exponentiation in a group of prime order—on the message symbols to produce a succinct digest of the file for verification. Hashes for code symbols can be calculated by multiplying together the hashes for the corresponding message symbols. Since the hashes for a file can be quite large—for example, an 8GB file would have 64MB of hashes—they also give a recursive hashing scheme that can greatly reduce the number of hashes sent over the wire. The security of the scheme reduces to the difficulty of computing discrete logarithms in the group. Their scheme has several "knobs" that allow adjusting the trade-offs between verification speed, amount of authentication information sent, and how quickly a malicious server can be identified (i.e., how many blocks must be received before maliciousness is detected). In comparing with our scheme, we note that we allow the adversary to have access to entire file and all of authentication tags while they assume reliable delivery of, at least, the root hash computed over an encoded file.

In [158], Tartary and Wang build on the work of Lysyanskaya et al. in [106], and use LT codes as part of a multicast authentication scheme over an (α, β)-network.[22] Specifically, they apply an LT code to a set of n packets and generate N code symbols. Each symbol is then hashed along with the indices of its neighbors. The hashes are signed and the signature is appended to their concatenation. The N hashes and signature are then encoded with an $[N, \alpha N]$ Reed-Solomon code. Each code symbol c_i from the LT code has the indices of its neighbors appended as well as the i-th code symbol of the encoded hashes and signature. To decode, they apply list decoding to encoded hashes and use the signature to achieve unique decoding (as in [106] and [111]) and recover the hashes which are used to detect corruption of the LT code symbols.

[21] If the client knows the correct Merkle tree root hash, then this partitioning and random selection is not necessary.

[22] Recall that an (α, β)-network is a network work, when sending n packets at least an αn packets will arrive unscathed, and at most βn will arrive total.

Network Coding. In [65], Ho et al. present an information-theoretically secure construction for random network coding where the adversary $\mathcal{A}$ is computationally-unbounded and controls a constant fraction of the network (but not all of it). Each packet is augmented with a polynomial hash of the packet, and then at each step in the network, all incoming packets are combined in a random linear combination. The decoder collects enough linearly independent packets to recover the input packets, checking the polynomial hashes against the decoded data and throwing out anything corrupted. Since $\mathcal{A}$ does not control the entire network, some of the packets are unknown to $\mathcal{A}$ and he is unable (except with small probability) to manipulate the packets to cause a decoding error. The probability of corruption detection is a tunable design parameter of the network itself (i.e., must be set before any transmissions).

3.9 Conclusion

In this work, we presented new concrete definitions for keyed coding schemes as well as a new adversarial model that is stronger than previously considered. We then provided two different constructions of keyed coding schemes: one public-key coding scheme combining list decoding with signatures and secure hash functions, and another private-key scheme combining message authentication codes with a pseudorandom permutation and a non-malleable cipher. The list-decoding-based construction achieves greater decoding efficiency than previous schemes (i.e., [106, 111]) by applying the list decoding to a small set of (short) authentication tags rather than the whole message, and hence can perform the decoding in a much smaller field. Also, our second construction avoids list decoding altogether and achieves greater throughput than the list decoding schemes when face with a high error rate. We also proved these schemes to be secure in our new model and with a detailed, concrete analysis we also demonstrated their practical efficiency.

CHAPTER **Four**

Falcon Codes

4.1 Introduction

By increasing the reliability of computing systems that may experience loss or corruption of data at rest or in transit, due to unreliable storage units or channels, error correcting codes comprise today a particularly useful tool that finds numerous applications in distributed systems and network security. Among a rich set of existing codes, due to their simplicity and strong error correcting capacity (i.e., information-theoretic rather than probabilistic), Reed-Solomon codes, or RS codes,[1] are employed widely in secure protocol design. Although their asymptotic efficiency depends on the implementation, RS codes typically involve encoding costs that are quadratic in the message size k,[2] thus in reality they tend to be costly.

Several alternative codes have been proposed to overcome the quadratic encoding/decoding overheads of RS codes. For instance, using layered encoding, Tornado codes [23] achieve encoding/decoding speeds that are 10^2 to 10^4 times faster than RS codes. At the fastest end of the range lie LT codes [103], which achieve $O(k \log k)$ encoding/decoding speed and are very practical. Based on LT codes, Raptor codes [148] and Online codes [108] are the first (rateless) code to achieve *linear* coding times. Each of these LT-based codes is a *rateless* (or *fountain*) code that can generate a practically unbounded stream of output code symbols. But this great efficiency comes at a qualitative

[1] In an RS code, the input message is broken up into fixed-sized pieces and the pieces are regarded as coefficients of a polynomial, which is then repeatedly evaluated on different points to produce the output symbols.

[2] RS codes with input size k and output size $n = O(k)$ are practically of quadratic in k complexity: even if in specific configurations they can achieve asymptotically $O(k \log k)$ encoding time, in most practical cases encoding with polynomial evaluation in $O(kn)$ time is faster.

drawback: fountain codes have been designed and analyzed over a *random (erasure) channel* rather than an adversarial (corruption) channel, essentially being capable to tolerate only random symbol *erasures* and no (or very limited) symbol corruption. And even current standardized implementations of these codes are easy to attack by adopting malicious (non-random) corruption strategies.

Vulnerabilities to Adversarial Corruptions. LT codes employ a random sparse bipartite graph to map message symbols, in one partition, into encoded symbols, in the other partition, via simple XORing (see Figure 4.1). By design, this graph provides enough coverage among symbols of the two partitions so that a belief-propagation decoding algorithm can recover the input symbols (with a small, encoder-determined probability of failure) despite random symbol erasures. But this algorithm will also readily *propagate (and amplify) any error* in the message encoding into the recovered message! This by itself is a serious problem as LT codes provide no mechanism for checking symbol integrity, thus, an attacker can trivially inflict a decoding failure or cause decoding errors that result in an incorrect recovered message (or even in a maliciously selected specific recovered message).

Even worse, an attacker can exploit the graph structure to (covertly) increase the likelihood of a decoding failure by only inflicting a few adversarial symbol erasures, in a class of attacks we term *targeted-erasure attacks*. Here, the attacker's goal is to maliciously select those symbol erasures that are more likely to cause decoding failure. For instance, one can selectively erase symbols of high-degree nodes in the encoding graph so that with high probability not all input symbols are sufficiently covered by the surviving encoded symbols.[3] Unfortunately, such targeted-erasure attacks have been neglected by existing RFCs (e.g., [101, 102]) that describe Raptor/RaptorQ codes for object delivery over the Internet: indeed, to increase the practicality of these codes, their encoding graph is either completely deterministic or easily predictable by anyone,[4] thus trivially enabling high-degree symbol (or other targeted) erasures! This raises a serious threat for real-life applications that (will) employ RaptorQ codes—the most advanced fountain code that is being adopted for protecting digital media broadcast, cellular networks, and satellite communications. Surprisingly, until very recently, such vulnerabilities had been inadequately addressed in the literature (cf. Section 4.8).

[3] LT decoders based on solving the implicit system of linear equations relating input symbols to encoded symbols suffer from the same problems.

[4] For instance, each encoded symbol contains a list of the indices of its covering input symbols or a seed for a PRG to generate these indices, trivially allowing an adversary to selectively corrupt the symbols for maximum impact. Encrypting symbols and the associated index lists is not sufficient as the RFCs contain explicit tables of random numbers to be used in the encoding process.

In the context of secure P2P storage, Krohn et al. [91] identified *distribution* (similar to targeted-erasure) attacks against Raptor-encoded data but their mitigation was left as an open problem. Recently, in the context of data transmission, Lopes and Neves identified [99] and successfully implemented [100] such an attack against RaptorQ codes, by relating encoded symbols to input symbols and deriving *data independent* erasure patterns that can increase the likelihood of decoding failure by *orders of magnitude*. Lopes and Neves [100] also informally proposed a basic remediation strategy that aims at making harder to discover input-output symbol associations by employing a cryptographically strong pseudorandom generator (PRG) for mapping encoded symbols to input nodes in the encoding graph. In Section 4.2, we explain why this strategy alone is inadequate to provide a secure solution,[5] and further justify the importance of (provable) secure rateless codes against fully adversarial corruptions both in terms of motivation and applications.

Then, in view of this inconvenient trade-off between practicality and security, in this chapter we consider the following natural question: is it possible to have codes that are simultaneously strongly tolerant to adversarial errors and very efficient in practice? Alternatively: what are the counterparts of RS codes within the class of fountain codes? Or, is it possible to devise extensions of Raptor codes that withstand malicious corruptions while maintaining their high efficiency?

Contributions. This work shows that it is possible to achieve both coding efficiency and strong tolerance of malicious errors for computationally-bounded adversaries. Specifically, we introduce *Falcon codes*, a class of *authenticated error correcting codes*[6] that are based on LT codes. Falcon codes can tolerate malicious symbol corruptions but also maintain very good performance, even *linear* encoding/decoding time. This can be viewed as a best-of-two-worlds quality, because existing authenticated codes (e.g., [21, 32, 88, 106, 111]) are typically Reed-Solomon based, thus lacking efficiency, and existing linear-time coding schemes (e.g., [108, 148]) are fountain codes that withstand only random erasures.

We develop the first adversarial model for analyzing the security of fountain codes against computationally-bounded adversaries.[7] We first introduce *private LT-coding schemes*, which model

[5] Intuitively, although the suggested approach holds some promise for tolerating *data independent* targeted erasures, it is inadequate to tolerate other types of targeted-erasure attacks, e.g., those that depend on *symbol contents*.

[6] By this term, here, we informally refer to error correcting codes that employ cryptography to withstand adversarial symbol corruptions, typically (but not exclusively) by authenticating the integrity of the encoded symbols.

[7] Previous adversarial models studied only fixed-rate codes, e.g., RS codes.

the abstraction of authenticated rateless codes that combine the structure of LT codes with secret-key cryptography, and we then define a security game in which a stateful and adaptive adversary inflicts corruptions over message encodings that aim at causing decoding errors or failures (i.e., a wrong message or no message is recovered).[8] Finally, we define security for private LT-coding schemes as the inability of the adversary to inflict corruptions that are (non-negligibly) more powerful than corruptions caused by a random erasure channel, i.e., security holds when any *adversarial (corruption) channel* is effectively reduced to the *random (erasure) channel*. Section 4.3 introduces technical background on relevant coding theory and cryptography, and Section 4.4 details our security formulation in our new adversarial setting applied to rateless codes.

We then provide three constructions of authenticated LT codes (which are our core Falcon codes) that achieve this new security notion against adversarial corruptions while preserving the efficiency of normal LT codes. Our main scheme Falcon (shown schematically in Figure 4.3) leverages a simple combination of a strong PRG (to randomize and protect the encoding graph), a semantically secure cipher (to encrypt encoded symbols and hide the encoding graph), and an unforgeable MAC (to authenticate symbols). By partitioning the input message into *blocks* and applying the main scheme in each block, we extend Falcon to get two scalable and highly optimized (but more elaborate in terms of parameterization and analysis) schemes: a fixed-rate code FalconS and its rateless extension FalconR that can produce unlimited encoded symbols.

Moreover, our core Falcon codes can be readily extended to meet the performance qualities of *any* other fountain code that employs an LT code. Specifically, our coding schemes can meet the performance optimality of Raptor or Online codes, while strictly improving their error correcting properties. In this view, Falcon codes provide a general design framework for devising authenticated error correcting codes, which overall renders them a useful general-purpose security tool. Section 4.5 details our design framework and three core Falcon codes, whereas Section 4.6 provides their security analysis.

In Section 4.7, we perform an extensive experimental evaluation of the Raptor-extensions of our core Falcon codes (since Raptor codes are perhaps the fastest linear-time fountain code at present) showing that, indeed, they achieve practical efficiency with low overhead. In particular, our schemes can achieve encoding and decoding speeds up to 300MB/s on a Core i5 processor, several times

[8] As we explain, we impose only minimal restrictions on the adversary: symbol corruptions and erasures can be arbitrary, but inducing trivial decoding failures by destroying (almost) all symbols is disallowed.

Construction	Rateless	Efficiency	Strong PRG	Weak PRG	Corruption
Falcon	yes	$O(k \log k)$	yes	no	yes
FalconS	no	$O(bk(\log k + \log b))$	yes	yes	yes
FalconR, small b	yes	$O(bk \log k)$	yes	no	yes
FalconR, large b	yes	$O(b \log b + bk \log \log b)$	yes	no	yes
Reed-Solomon	no	$O(nk)$	n/a	n/a	yes
LT code & crypto	yes	$O(k \log k)$	no	yes	no

Table 4.1: Comparison of three Falcon code variants with Reed-Solomon and LT codes. Here, k is message length, b is number of blocks, and n is the codeword length. Falcon is the basic scheme and FalconS and FalconR are the scalable variants. For FalconS and FalconR, k indicates the number of symbols per block.

faster than the standard RS encoding coupled with encryption and MACs. The overhead from our cryptographic additions to LT codes results in a slowdown of no more than 60% with typical slowdown close to just 25%. Table 4.1 compares our constructions with a standard RS code, as well as the naïvely "secure" LT code also coupled with encrypts and MACs the output symbols, according to the following criteria: (1) whether or not the code is rateless, (2) asymptotic efficiency of encoding and decoding (FalconR's performance depends on the number of blocks, as we discuss in Section 4.5.3), (3) the need for a strong PRG (FalconS can safely employ a weak, but fast, PRG for increased efficiency), (4) and the achieved security related to adversarial data corruption. All of our constructions can withstand this worst-case behavior.

Applying Falcon codes to reliable data transmission and storage is out of the scope of this work, but we believe that many such applications are feasible. In particular, as we briefly discuss in Section 4.2, our schemes can be used as drop-in replacements for any error correcting code used against computationally-bounded adversaries to provide immediate efficiency gains. Finally, we overview related work in the overlap of coding and security in Section 4.8 and conclude with Section 4.9.

4.2 Motivation & Applications

Our goal is this work is to design *fast, authenticated (rateless) codes* that offer both strong tolerance to adversarial symbol corruptions and operate at practical encoding rates. This problem lies in the intersection of security and coding and is motivated by an interesting trade-off between security and efficiency that we can observe in two wide application areas:

- **Distributed or cloud storage systems:** For better fault-tolerance guarantees against server failures or security guarantees against malicious cloud providers, several storage systems employ (adversarial) error correcting codes, which are typically RS codes, thus protecting against malicious corruptions of data at rest but often being less practical.
- **Data transmission systems:** For fast data recovery against lossy channels, several data transmission systems employ linear-time fountain (rateless) codes, thus being very practical but protecting data in transit only against random symbol erasures.

To reconcile the above desired properties—strong resilience to adversarial corruptions and fast, linear time, encoding of data both at rest and in transit—we study rateless codes, in particular LT codes and Raptor codes, in a new adversarial model where an attacker has *full control* over the message encoding. This means that the attacker can *maliciously corrupt* (i.e., alter or erase) symbols in any subset of its choice in a message encoding. In this case, we capture security by requiring that any attacker in this setting is essentially equivalent to a much more limited attacker in the weaker setting where only random symbol erasures are allowed. That is, using some cryptographic terminology, *the real-model adversarial channel is reduced to the ideal-model random channel.*

Therefore, we define *private LT-coding schemes* as the secure equivalents of LT codes in this new adversarial model, implying that private LT-coding schemes (and Falcon codes), when operating over an adversarial channel, inherit all the properties of LT codes operating over a random channel. Most importantly, as we discuss in Section 4.3, LT codes assume a minimum number $(1+\varepsilon)k$ of *randomly-selected intact* symbols in the encoding of a k-symbol message in order to guarantee successful decoding with probability at least $1-\delta$ (where $\varepsilon > 0$ controls the redundancy required to ensure the decoding-failure probability it at most $\delta > 0$, where ε depends on δ). We, thus, have to assume that the same precondition for successful decodings holds also for our private LT-coding schemes. Namely, given any minimum number of $(1+\varepsilon)k$ (but this time) *adversarially-selected intact*, along with any number of *adversarially-altered*, encoded symbols, the attacker cannot induce any decoding failure or error, except with probability at most δ.[9] Similarly, an attacker is not allowed to corrupt or erase all encoded symbols—which would trivially incur a decoding error—because such a contrived attack is already inconsistent in the ideal-model random channel.[10]

[9] Note, however, the qualitative difference between (standard) LT codes and Falcon codes: in both cases there exist some "bad" subsets of $(1+\varepsilon)k$ intact encoded symbols that produce decoding failures; but with high probability such "bad" subsets cannot be computed by an attacker from a Falcon encoding, whereas they can almost always be computed from standard LT encodings, as we discuss below.

[10] Indeed, this attack is practically infeasible in a channel that independently erases symbols with probability $p < 1$.

In practice, there are several adversarial settings that match (and justify) our security model above. First and foremost, in any application, we want to ensure that, with all but negligible probability, whenever the decoder outputs a message, it is the same message that was encoded. That is, we wish to avoid corruption of the decoded message.[11] Then, since total corruption or deletion of a message is catastrophic—and presumably users would stop using such an unreliable channel—it is realistic to consider attacks that do not destroy or maul the entire message (which is also a standard assumption when analyzing error correcting codes). In storage systems, for instance, client data may be encoded and distributed among several cloud providers, a subset of which are compromised by the adversary, limiting corruption to those servers. Or, in transmission systems, the adversary may be a malicious network router that sees some, but not all of the symbols in a message encoding, which it can arbitrarily corrupt. Finally, we want to protect against targeted-erasure attacks, which are minimalistic attacks where the attacker maliciously erases a carefully selected subset of the encoded symbols to significantly increase the chances of a decoding failure way above the bound of δ. In the case above of the malicious router, for instance, the attacker may erase all high-degree symbols (i.e., those combining many input symbols), leaving only low-degree symbols untouched, causing the decoder to then receive mostly low-degree symbols and be much more likely (i.e., with probability greater than δ) to fail and output nothing. This is a stealthy DoS, or at least a quality-of-service attack, and is a violation of the upper-bound on the decoding failure probability. To mitigate this, the receiver could wait for more good symbols from other non-malicious routers, but this would still break the LT code's guarantee of needing only $(1+\varepsilon)k$ (randomly sampled) uncorrupted symbols to decode with probability greater than $1-\delta$. We wish security even from such low-profile targeted-erasure attacks, especially because they are feasible: there are easy to perform and they are very effective.

Indeed, the work of Lopes and Neves [100] illustrates the practicality of the targeted-erasure attack against RaptorQ codes. They demonstrate *data independent* attacks on encoded data such that the probability of decoding failure is increased by several orders of magnitude. The attack is as follows. RaptorQ codes include unique identifiers with each symbol to allow easy reconstruction of the encoding graph by the decoder. Each message symbol is associated to an "Internal Symbol ID" (ISI) label and each parity symbol to an "Encoded Symbol ID" (ESI) label. (RaptorQ is a systematic code, so some ISI and ESI labels will coincide.) RaptorQ codes generate a parity symbol's degree

[11] Note that errors propagate and amplify for a belief-propagation LT decoder: When the value of a message symbol is fixed, it is propagated to every encoded symbol associated to that message symbol, setting each of these to erroneous values, which can then lead to the corruption of other symbols as well.

and associated neighbors by feeding the ISI and the message length k into a "tuple generator" which utilizes a (very) weak PRG (utilizing a fixed table of random values). The attack leverages the fact that given the ESI it is possible to derive the ISI and thereby determine how a given parity symbol was encoded. Then, a brute-force algorithm is used to find minimal erasure patterns for selectively deleting symbols to ensure (with some probability) that decoding fails.

Lopes and Neves [100] also informally suggest (without any correctness or security arguments) some possible mitigations, based on an observation that simply using a strong pseudorandom generator for the encoding is sufficient to protect against malicious erasures. Their suggested mitigations include to: (M1) encrypt and permute the symbols; (M2) give each ISI a random ESI and then randomly "interleave" the symbols; or (M3) give each ESI a random ISI (and do not permute). Although (probably) sufficient to mitigate the specific attack in [100], the above strategies seem insufficient to provide a viable and secure solution in the design of authenticated LT codes, for the following reasons: (R1) they rely on a random permutation (or random interleaving) that undermines ratelessness; (R2) they only hide the input to the weak PRG (tuple generator), but they do not modify the PRG itself whose biased output can still be exploited to predict the encoding graph structure; and (R3) they do not protect against data-dependent attacks (i.e., inferring graph structure from message contents).

In particular, we next show that it is possible for an attacker to look at the symbols to infer the encoding graph structure and again enact the targeted-erasure attack. While this is hard in general, for structured data or attacker-known input messages, the low-degree symbols can be readily identified. Symbols of degree 1 are trivial to identify in transit and, with a known input message, the attacker can take all pairs of message symbols and compute all degree 2 symbols.[12] Then with this knowledge, the attacker can perform three different attacks:

- A1 Identify the low-degree symbols and delete all others; with high probability these low-degree symbols will not provide enough coverage of the input symbols and decoding will fail;
- A2 If the receiver is known to use a belief-propagation decoder, the attacker only needs to delete the symbols of degree 1 to prevent decoding from even starting;[13]
- A3 If one of the input symbols is distinguished (e.g., the only one with the MSB set), then it is trivial to identify which code symbols contain that input symbol; deleting all symbols with that bit set guarantees that decoding will fail.[14]

[12] If the input is small enough, the attacker may even be able to compute all symbols of degree 3.

[13] However, a decoder based on Gaussian elimination to solve the system of linear equations that describe the encoding graph would likely still succeed.

[14] As an example, consider an input message consisting of at most 256 ASCII characters with the length prepended

Thus, we must provide additional protections (namely encryption) to prevent these data-dependent attacks. Overall, given the many subtleties in such complicated structures, a formal proof of security is required for any suggested solution.

Overall, our Falcon codes comprise private LT-coding schemes, thus realizing fast, authenticated rateless codes that achieve both desired properties described at the beginning of the section. As such, they can be used as drop-in replacements for any error correcting code used against a computationally-bounded adversary and provide immediate efficiency gains. For instance, Falcon codes can replace RS codes to provide fault-tolerance in secure storage with almost the same guarantees as RS codes but much higher efficiency. They also readily find application in PoR systems that provide auditing checks on the retrievability of cloud data—we provide some more details about this application in Appendix B. Similarly, Falcon codes can be applied to secure against corruptions in any application where Raptor codes are used (see, e.g., [136]), including 3GPP, MBMS, streaming media [1], IP TV, IP Datacast over DVB-H [133], or satellite communications [132].

4.3 Preliminaries

In what follows, we let λ denote the security parameter, and PPT refer to probabilistic polynomial-time algorithms. We also let $[\mathsf{Alg}]$ denote the set of all possible outputs of a PPT algorithm Alg, running on input parameters π, and $\tau \leftarrow \mathsf{Alg}(\pi)$ the particular output derived by a specific random execution of $\mathsf{Alg}(\pi)$. Analogously, we let $x \xleftarrow{R} S$ and $x \leftarrow D$, respectively, denote the process of sampling x from the set S uniformly at random or according to distribution D. We have (a, b) denote the open interval from a to b. Finally, we let $\circ$ denote string concatenation and $|S|$ denote the cardinality of a set S.

4.3.1 Coding Theory

An *error correcting code* (ECC) is a message encoding scheme that can tolerate some corruption of the encoded data and still allow recovery of the message from the (possibly corrupted) codeword. Codes that are designed to recover only from partial data loss, but not data corruption, are called *erasure codes*. We first present the definition of *fixed rate* codes (also called *block* codes) which are the most common type of ECCs. Later, we will define *rateless* codes.

to the message. If the length is ≥ 128 characters, then all symbols that include the length byte are easily identifiable.

Defined over a fixed, finite set of symbols Σ (called the *alphabet*) and parameterized by integers k and $n \geq k$ (called the *message length* and *block length*, respectively), a fixed-rate ECC specifies mappings between elements of the set of *messages* Σ^k and the set of *codewords* Σ^n. (Examples alphabets include $\Sigma = \{0,1\}^l$, the set of all l-bit strings, or a finite field $\mathbb{F}$.) The ratio $R = k/n$ is the *rate* of the code, capturing the amount of information transmitted per codeword. For $m \in \Sigma^k$ and $c \in \Sigma^n$, the components m_i and c_i are called *message symbols* and *code symbols*, respectively. Any given message $m \in \Sigma^k$ is *encoded* to a corresponding *valid* codeword $c \in \Sigma^n$, and any *invalid* codeword $\hat{c}$, derived as a bounded distortion of c, can be *uniquely* decoded back to the original message m. If the first k symbols of a valid codeword corresponds to original k message symbols, then the code is said to be *systematic*. The rate R of a code typically controls the recovery strength of the code as follows. The *Hamming distance* between two n-symbol words x and y is the number of symbols in which they differ, defined as $\Delta(x,y) = |\{i \mid 1 \leq i \leq n, x_i \neq y_i\}|$. The *(minimum) distance* of an ECC is d, if for all valid codewords $c, c' \in \Sigma^n$ such that $c \neq c'$, we have $\Delta(c,c') \geq d$ (capturing the minimum number of changes needed to transform one valid codeword into another). Then, any code with minimum distance d allows for decoding invalid codewords distorted by up to $\lfloor d/2 \rfloor$ errors back to a unique message;[15] typically, smaller values of R imply a larger minimum distance d.

Definition 4.1. [Error Correcting Code]An *error correcting code* C over an alphabet Σ with rate R and minimum distance d, is a pair of maps $(\mathsf{Encode}, \mathsf{Decode})$, where $\mathsf{Encode} : \Sigma^k \to \Sigma^n$ and $\mathsf{Decode} : \Sigma^n \to \Sigma^k$, such that $k = Rn$, and for all $m \in \Sigma^k$ and for all $c \in \Sigma^n$ with $\Delta(c, \mathsf{Encode}(m)) \leq \lfloor d/2 \rfloor$, $\mathsf{Decode}(c) = m$.

Rateless ECCs. Error correcting codes that employ no fixed block length n are called called *rateless* or *fountain* codes. These codes can generate an unbounded stream of encoded symbols (i.e., a continuous "fountain"), allowing recovery of the original message (with high probability) from any random subset of these symbols that is sufficiently large. Typically, for message length k, $(1+\varepsilon)k$ correct encoded symbols are needed to decode the message with probability at least $1-\delta$. Here, ε is called the *overhead* of the code and δ refers to the *decoding failure probability* of the code, the latter determining the range of possible values that the former may have; in particular, smaller values of δ require larger values of ε and vice versa. We denote this relationship by $F(\delta, \varepsilon)$.

[15] Not considered in this work is list-decoding, which allows mapping any invalid codeword, distorted with errors *beyond* this half-the-distance bound, back to a list of messages that always contains the correct original message.

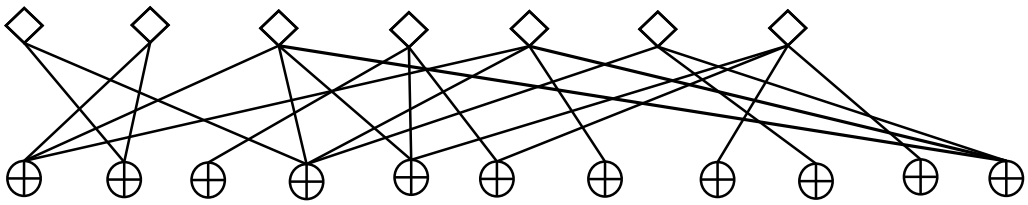


Figure 4.1: LT-encoding: Each code symbol (bottom) is the XOR of $O(\log k)$ randomly selected message symbols (top).

Rateless codes operate over the *random erasure channel*, where each symbol is independently erased with some fixed probability p, and it is relative to this channel that its probabilistic guarantees hold.[16] We denote the channel by REC_p. (Note, this channel can be generalized so that the value of p varies per-symbol and can even depend on the symbols previously sent.) One desirable feature of rateless codes is that the value of p does not need to be known to the encoder a priori.

Definition 4.2. A (k, δ, ε)-*rateless error correcting code* C over an alphabet Σ with decoding failure probability δ and overhead ε so that $F(\delta, \varepsilon)$, is a pair of maps $(\mathsf{Encode}, \mathsf{Decode})$, where Encode maps elements of Σ^k to infinite sequences $\{c_i\}_{i=1}^{\infty}$, with $c_i \in \Sigma$, and for any $m \in \Sigma^k$ and any finite subsequence s of $\mathsf{Encode}(m)$ of length at least $(1+\varepsilon)k$ received over REC_p for some $p \in (0,1)$, $\mathsf{Decode}(s) = m$ with probability at least $1 - \delta$.

Examples of rateless ECCs include LT codes [103], along with their derived extensions Raptor codes [148] and Online codes [108]. For LT codes, the main focus in this work, the encoding/decoding mappings take an extra input parameter: the *degree distribution* $\mathcal{D}$. Here, $\mathcal{D}$ is used to construct a sparse bipartite graph with input (message) symbols in one partition and code symbols in the other, also called *input* and *parity nodes*—see Figure 4.1. The degree of each parity node is selected according to $\mathcal{D}$ and its corresponding neighbors (message symbols) are selected uniformly at random. The code symbol corresponding to a given parity node is simply the XOR of the message symbols of the neighboring input nodes (and, thus, is very fast to compute). The distribution used must be carefully chosen to achieve the desired success probability of $1 - \delta$ for a given message length k, hence $\mathcal{D}$ is implicitly parameterized by k and δ. (For notational simplicity, we will leave this parameterization implicit.) The value of δ and, now also, distribution $\mathcal{D}$ determine the possible values of ε. For LT codes, we denote this relationship by $F(\delta, \mathcal{D}, \varepsilon)$.

[16] However, as shown in [100], these guarantees can fail to hold when the channel is malicious.

Often, $\mathcal{D}$ instantiates to the *robust soliton distribution*, described in the original LT Code paper [103], which ensures that the average node degree is $O(\log k)$. This implies that: (1) using a balls-in-bins analysis, with high probability, every input symbol is covered by at least one of the $(1+\varepsilon)k$ code symbols, and thus can be decoded; and (2) encoding and decoding take $O(k \log k)$ time. Decoding uses a standard belief-propagation algorithm,[17] thus correcting only symbol *erasures*, but not symbol *errors*. The decoding method of [104] augments belief propagation to also correct *random* errors with erroneous values distributed uniformly in Σ, but *not* adversarial errors.

Raptor (*rap*id *tor*nado) codes [148], a main application in our work, improve the performance of LT codes by employing *precoding* of the input message $m \in \Sigma^k$ before LT-encoding as follows. First, a linear-time erasure code (e.g., a low-density parity check code) is applied to m to get a group of *intermediate symbols*. Then, an LT code is used to produce each output symbol, again as the XOR of a random subset of the intermediate symbols, where these subsets, drawn via a variant of the robust soliton distribution (see [148]), are of *constant size*; this process is repeated until enough output symbols are produced. Overall, Raptor codes have encoding/decoding time that is linear in k, and achieve high data rates with low overhead.[18] However, based on LT codes, they are essentially erasure codes and do not tolerate symbol corruption well. Analysis of Raptor codes over noisy channels (see [120, 129]) is also restricted to random (but not necessarily uniform) errors rather than adversarial ones.

4.3.2 Cryptographic Tools

We overview the cryptographic primitives we employ to enhance LT codes to withstand adversarial errors—namely, message authentication codes (MACs), symmetric ciphers (SCs), and pseudorandom generators (PRGs), with which basic familiarity is assumed.

Keyed by secret $sk \leftarrow \mathsf{Gen}_1(1^\lambda)$, a MAC produces a tag $t = \mathsf{Mac}(sk, m)$ for message m, which can be used to verify the integrity of m by checking $\mathsf{VerifyMac}(sk, m, t) = 1$. We require that a MAC is *existentially unforgeable* so that an adversary $\mathcal{A}$ cannot forge the tag for *any* message, even a message of its own choosing. Here, $\mathcal{A}$ is allowed to query a MAC oracle to receive any number of example

[17] Briefly, a belief-propagation decoding algorithm works by searching for a code symbol that has degree 1 and then setting the corresponding message symbol to the value of the code symbol. The message symbol is then XORed with each of its neighboring code symbols, decrementing each of their degrees by one. This process repeats until there are no more code symbols of degree one. Then, the decoder processes the next received code symbol. If there are no more code symbols and there are still undetermined message symbols, then decoding fails.

[18] Since only a linear number of symbols are output, Raptor codes only strive to recover a *constant fraction* of the intermediate symbols via LT-decoding. Any gaps in these symbols are recovered by decoding the precode.

message-tag pairs before it outputs a target message-tag pair (m^*, τ^*) that does not belong in the queried pairs; then, with all but negligible probability in λ it holds that $\mathsf{VerifyMac}(sk, m^*, t^*) \neq 1$.

Keyed by secret $sk \leftarrow \mathsf{Gen}_2(1^\lambda)$, a symmetric cipher is an encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ so that $\mathsf{Dec}(sk, \mathsf{Enc}(sk, m)) = m$ for any message m (in the appropriate message space). We require that an SC is *semantically secure* so that a ciphertext $c = \mathsf{Enc}(sk, m)$ "hides" all the information about a given message m. Here, for any adversary $\mathcal{A}$ computing any function f on the message m given ciphertext c (and any auxiliary input), $f(m)$ can still be computed *without* the ciphertext, with all but negligible probability in λ. Intuitively, knowing the ciphertext c leaks no additional information about m.

Finally, given a short random seed $s \xleftarrow{R} \{0,1\}^\lambda$, a pseudorandom generator (PRG) serves as an efficient source of randomness by producing a long sequence of random-looking bits. A PRG is secure if its output is indistinguishable, with all but negligible probability in λ and with respect to any polynomial-time distinguisher, from a string of truly random bits. Equivalently, any algorithm taking random bits as input behaves only negligibly different when given pseudorandom bits instead.

4.4 Security Model

Our main goal is to extend LT codes to endure *adversarial* corruptions inflicted by a computationally-bounded adversary. Here, we present a new definitional framework for *private LT-coding schemes*, a new class of rateless codes that are based on LT codes and employ the use of secret-key[19] cryptography to resist polynomial-time adversarial errors. We introduce a corresponding new security notion we call *computationally secure rateless coding*. Our security model is general enough to also capture security for block codes.

Motivating Scenarios. There are several adversarial settings which motivate our security model. First and foremost, we want to ensure that, with all but negligible probability, whenever Decode outputs a message, it is the same message that was encoded. That is, we wish to avoid corruption of the decoded message. Since total corruption or deletion of a message is catastrophic—and presumably users would stop using such an unreliable channel—we consider attacks that do not destroy or maul

[19] Our schemes use a PRG, a MAC, and a semantically secure cipher: the latter two can be replaced with public-key equivalents, while the PRG is inherently a secret-key scheme. However, coupling our schemes with a public-key key-agreement protocol (over a noiseless channel) for distribution of the PRG seed would break the dependence on secret-key cryptography. But, for simplicity of analysis and presentation, we only utilize secret-key schemes.

the entire message, which is standard when analyzing ECCs. For example, a client may encode their data and distribute it among several cloud providers, a subset of which are compromised by the adversary $\mathcal{A}$, limiting corruption to those servers.

An alternative scenario is where $\mathcal{A}$ is malicious network router that sees some, but not all of an encoded message. $\mathcal{A}$ can attack an LT-encoded message by erasing high-degree symbols (i.e., those combining many input symbols) and leaving low-degree symbols untouched. The decoder would then receive mostly low-degree symbols and be much more likely (i.e., with probability greater than δ) to fail and output nothing (see [100] for an implementation of this attack). This is a stealthy DoS, or at least a quality-of-service attack, and is a violation of the upper-bound on the decoding failure probability. To mitigate this, the receiver could wait for more good symbols from the non-malicious routers, but this would break the LT code's guarantee of needing only $(1+\varepsilon)k$ uncorrupted symbols to decode with probability greater than $1-\delta$. We seek to be secure even from this *targeted-erasure* attack, an attack which was previously identified by Krohn et al. in [91] (and called a *distribution attack*), but was left as an open problem and only partially mitigated by Lopes and Neves in [100] (see Section 4.2).

4.4.1 Private LT-coding Schemes

Prior work formalizing computationally-bounded adversaries *only* considered block codes. In particular, in his seminal paper [98], Lipton first modeled a *computationally-bounded adversarial channel* that can corrupt at most a constant fraction ρ of the encoded symbols. Later, Lysyanskaya et al. [106] studied an (α, β)-*network (or channel)* that can arbitrarily corrupt (alter or delete) the n code symbols in an encoded message, and also insert new symbols into the codeword (even multiple versions of a symbol), subject to two restrictions: (1) at least αn symbols survive corruption and (2) at most βn total symbols are received. Subsequent work by Micali et al. [111] provided a more involved security game that includes several rounds of encoding, bounded symbol corruptions, and decoding between a sender, an adversarial channel, and a receiver.

Bowers et al. [21] define an *adversarial ECC* which is a code that can withstand some amount of adversarial corruption of the codeword. Their adversarial channel is described via a game where the adversary $\mathcal{A}$ is given access to Encode and Decode oracles and seeks to output two codewords c and c'that decode to different messages while minimizing $\Delta(c, c')$. However, their game is intrinsically tied to block codes and does not generalized to rateless codes since it measures (and bounds) the

number of changes $\mathcal{A}$ must perform to have c and c' decode to different messages, which is potentially unbounded for rateless codes.[20]

By explicitly bounding the fraction of all symbols that can be corrupted, however, these security models cannot capture corruptions against rateless codes. Since fountain codes can produce an unbounded number of symbols, the rate of corruption introduced by the adversary can continually grow and, indeed, can become *arbitrarily close* to 1, which would directly conflict with any bounded corruption rates. A more accurate modeling of errors over a rateless code is to *lower bound* the amount of *non-corruption* rather than upper-bound the amount of corruption, but as an absolute number (typically defined by the message length), not as a fraction of the encoded symbols. That is, we wish to ensure that there is some *minimum number* of "good" encoded symbols that remain intact, and allow the remainder to be bad. In a block code, an upper-bound on badness implies a lower-bound on goodness (and vice versa), but this symmetry breaks in rateless codes.

We first provide the definition of *private LT-coding schemes*. This extends the definition of private coding schemes given in [111], via changes in the parameters and function specifications, to suit the class of rateless codes that are based on LT codes (instead of block codes). This class generically captures fountain codes that can produce an unlimited number of encoded symbols using the LT-coding technique over a set of "input" symbols (not necessarily the message symbols) using an appropriate degree distribution $\mathcal{D}$ (thus encompassing LT codes, Raptor codes and Online codes). Recall that for a degree distribution $\mathcal{D}$ for a message length k, we want that, given $(1+\varepsilon)k$ code symbols, decoding succeeds with probability $1-\delta$. This relationship is denoted by $F(\delta, \mathcal{D}, \varepsilon)$. As in [98], we assume that the sender and receiver have a shared secret key sk. We also use nonces to prevent replays.

Definition 4.3. [Private LT-Coding Scheme] A $(k, \delta, \mathcal{D}, \varepsilon)$-*private LT-coding scheme* over an alphabet Σ with message length k, decoding failure probability δ, overhead ε and degree distribution $\mathcal{D}$ such that $F(\delta, \mathcal{D}, \varepsilon)$, and key space $\mathcal{K}$, is a triple of PPT algorithms $(\mathsf{Gen}, \mathsf{Encode}, \mathsf{Decode})$, where:

- Gen: on input security parameter 1^λ, outputs a random *secret key* $sk \in \mathcal{K}$;
- Encode: on input (1) secret key sk, (2) nonce ℓ, (3) decoding failure probability δ, (4) degree distribution $\mathcal{D}$, (5) overhead ε, (6) and the message $m \in \Sigma^k$, outputs an infinite sequence $\{c_i\}_{i=1}^{\infty}$, with $c_i \in \Sigma$, referred to as a *codeword* or an *encoding* of m;

[20] The game also explicitly disallows decoding failure from being an adversarial victory and so cannot model the targeted-erasure attack.

- Decode: on input (1) secret key sk, (2) nonce ℓ, (3) decoding failure probability δ, (4) degree distribution $\mathcal{D}$, (5) overhead ε, (6) and a string $c \in \Sigma^*$, where $|c| \geq (1+\varepsilon)k$, outputs a string $m' \in \Sigma^k$ or fails and outputs $\perp$.

We require that for all $m \in \Sigma^k$, $\mathsf{Decode}(sk, \ell, \delta, \mathcal{D}, \varepsilon, c) = m$ with probability at least $1-\delta$, where c is a finite subsequence of $\mathsf{Encode}(sk, \ell, \delta, \mathcal{D}, \varepsilon, m)$, of length at least $(1+\varepsilon)k$, received over REC_p for some $p \in (0,1)$.

The above definition is general enough to also express two types of "crypto-enabled" block codes. First, any ECC with fixed rate ρ can be captured by having a null distribution $\mathcal{D}$, if necessary, and adjusting δ and ε according to the code (e.g., for Reed-Solomon codes $\delta = \varepsilon = 0$, while for Tornado codes $\delta, \varepsilon > 0$). More importantly, we can refine Definition 4.3 to get *block* versions of LT-coding schemes that simply produce codewords of *fixed size* (above the decoding threshold). A $(k, \delta, \mathcal{D}, \varepsilon)$-*private block LT-coding scheme* with block-size n is defined as a $(k, \delta, \mathcal{D}, \varepsilon)$-private LT-coding scheme where Encode, given an additional input parameter n, produces codewords c with $|c| = n \geq (1+\varepsilon)k$. In what follows, let $\mathcal{LTS} = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{Decode}, \pi)$, denote a $(k, \delta, \mathcal{D}, \varepsilon)$-private LT-coding scheme with $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$, its n-symbol refinement by $\mathcal{LTS}_n = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{Decode}, \pi, n)$, and any (rateless or block) LT-coding scheme by $\mathcal{LTS}_*$.

Also, note that the probabilistic decoding requirement expressed by relation $F(\delta, \mathcal{D}, \varepsilon)$ imposes a minimum expansion factor $(1+\varepsilon)$ on message encoding. However, due to its dependence on the distribution $\mathcal{D}$, the failure bound δ holds when there are "enough" code symbols produced in an absolute sense. In practice, this further restricts message length k to be sufficiently large.[21]

4.4.2 Security Game

We next present our general security model against computationally-bounded adversaries that is applicable to private LT-coding schemes (and their fixed-size-output variants). Our goal is to capture tolerance against adversarial symbol corruptions for LT-coding schemes, thus defining a much stronger security notion over the existing one that considers only random symbol corruptions and erasures.

We thus consider a transmission channel that is fully controlled by a computationally-bounded adversary $\mathcal{A}$. That is, as new code symbols are produced by a private LT-coding scheme $\mathcal{LTS}$, $\mathcal{A}$ can maliciously corrupt any new or past such symbols. Moreover, the adversary is allowed to *adaptively*

[21] For example, the distribution used in Raptor codes works well when k is in the tens of thousands or greater (see [148]). Distributions for smaller k (e.g., in the thousands) must be carefully designed, see Section VII of [148].

interact with the channel. That is, $\mathcal{A}$ is allowed to examine *any symbols* of its choice of the encoding (produced by $\mathcal{LTS}$) of *any message* of its choice and, additionally, to examine the decoded message (produced by $\mathcal{LTS}$) on *any corrupted set of symbols* of its choice. Finally, we consider a *stateful* channel where $\mathcal{A}$ can remember any past selected message, its encoding, symbol corruptions and the corresponding recovered message, and depend current actions on the full such past history. (Do note, however, that $\mathcal{A}$ is never given the bipartite graph underlying any of the LT-encodings.) We call adversaries that do not remember prior rounds of encoding and decoding *stateless* or *memoryless*.

We define security in terms of a game $\mathsf{ChannelExp}_{\mathcal{A},\mathcal{LTS}_*}(\pi)$, shown in Figure 4.2, that the adversary $\mathcal{A}$ seeks to win. $\mathcal{A}$ wins by either: (1) causing a *decoding failure*, where Decode outputs $\perp$; or (2) by causing a *decoding error*, where Decode outputs a message different from the one originally encoded. There are three participants in the game: the encoder Encode and decoder Decode of a given LT-coding scheme $\mathcal{LTS}_*$ with parameters $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$, and the adversary $\mathcal{A}$. The game consists of a "learning" phase and an "attack" phase. In the learning phase, there is a sequence of at most a polynomial number of rounds where, in the i-th round: (1) $\mathcal{A}$ selects a message m_i to be encoded by Encode; (2) Encode initializes itself with m_i; (3) $\mathcal{A}$ queries different symbols from Encode by having oracle access to symbols of the encoding of m_i; in particular, $\mathcal{A}$ interacts with oracle $\mathcal{O}_{m_i}$ which given as input an index j, it returns the j-th encoded symbol produced by Encode (for block codes, if $j > n$ then $\mathcal{O}_{m_i}(j)$ outputs $\perp$); (4) $\mathcal{A}$ outputs a (corrupted) codeword c_i consisting of N_i symbols in $\Sigma \cup \{\perp\}$, where a symbol $\sigma_i = \perp$ if it has been erased; (5) c_i is given to Decode for decoding; (6) Decode outputs a message r_i that is returned to $\mathcal{A}$. For simplicity, we assume that $\mathcal{A}$ outputs at least $(1+\varepsilon)k$ symbols, good or bad, at each step (otherwise Decode trivially fails and outputs $\perp$). Memoryless adversaries skip the learning phase.

Eventually, $\mathcal{A}$ decides to enter the attack phase against scheme $\mathcal{LTS}_*$. The game proceeds with a special final round: $\mathcal{A}$ selects attack message m_a, queries encoded symbols of m_a using the oracle $\mathcal{O}_{m_a}(\cdot)$, and tries to cause decoding failure or a decoding error by computing a corrupted codeword c_a that decodes into message r_a. If Decode failed (i.e., $r_a = \perp$) or decoded the wrong message (i.e., $r_a \neq m_a$), then $\mathcal{A}$ wins; else, $\mathcal{A}$ loses. However, we require that, in the case of decoding failure, $\mathcal{A}$ must output at least $(1+\varepsilon)k$ unerased and uncorrupted symbols to win. Otherwise, $\mathcal{A}$ can trivially cause Decode to fail simply outputting only corrupted symbols, or by erasing almost all symbols.[22]

[22]Note, forged symbols are superfluous when trying to cause decoding failure since if a forgery causes a decoding failure, then the corresponding *uncorrupted* symbol would have caused the same failure.

For a decoding error, we have no such restriction. Let Q_m denote the set of symbols queried by $\mathcal{A}$ from $\mathcal{O}_m$. Abusing notation, let $c \cap Q_m$ denote the subset of symbols in codeword c (which was output by $\mathcal{A}$) that are in Q_m. Thus, for decoding failure, we require that $|c_a \cap Q_{m_a}| \geq (1+\varepsilon)k$.

The game ChannelExp that we define here is similar to the game (of the same name) defined in Chapter 3 Section 3.4. In that game, the adversary $\mathcal{A}$ goes through several rounds of submitting messages for encoding, attacking the encoded messages, and then seeing the results of the attacks. $\mathcal{A}$ wins that game if *any* round results in a decoding *error*, but not a decoding failure (which was modeled on the security game for ECCs in [111]). Here, we provide a novel security game and in it we divide the adversary's actions into two phases: a learning phase and an attack phase, putting the game in line with typical cryptographic-style games (e.g., the game for ciphertext indistinguishability). ChannelExp in this chapter models adversarial interactions with *rateless* codes while the game in Chapter 3 is intended to model attacks on *fixed-rate* (e.g., by giving the entire encoded message to $\mathcal{A}$, which is not possible with rateless codes). Indeed, the game in Figure 4.2 allows for the adversary $\mathcal{A}$ to win by causing a decoding failure, e.g., via a targeted-erasure attack (an attack that is not possible on a fixed-rate code) while decoding failures are explicitly *disallowed* for $\mathcal{A}$ to win in Section 3.4.

As a technical note, we disallow $\mathcal{A}$ from querying symbols that are arbitrarily far along in the stream produced by $\mathcal{LTS}_*$ on any input message, thus avoiding the situation where $\mathcal{A}$ queries symbols that are infeasible for Encode to produce sequentially—this could give $\mathcal{A}$ an unrealistic amount of power.[23] In the above game, we call a query $\mathcal{O}_m(i)$ for the i-th symbol (τ, p)-*feasible* if $Q_p(i) \geq \tau$, where $Q_p(i) = P(\text{Encode outputs at least } i \text{ symbols over } REC_p)$, and, further, we call τ-*admissible* any $\mathcal{A}$ making only τ-feasible queries for a given p.

Note, we must also restrict p so that $1-p$ is a non-negligible function of λ (we call such values of p *feasible*) to ensure that a non-negligible fraction of the code symbols are expected to survive transmission (i.e., we assume it is feasible to communicate over the channel). If p is permitted to be negligibly close to 1, then $Q_p(i)$ is non-negligible for super-polynomial values of i. For a block code with message length k and block length n, we assume that $p \in (0, 1-k/n)$ and define p to be feasible if it is non-negligibly different from $1-k/n$. If $\mathcal{A}$ is τ-admissible for all feasible p and, additionally, τ is also non-negligible in λ, then we call $\mathcal{A}$ *admissible*. In what follows, we consider only admissible adversaries and feasible p.

[23] If $\mathcal{LTS}_*$ employs a PRG with finite state, $\mathcal{A}$ can simply query symbols at multiples of the period of the PRG, thus getting identically encoded symbols (as they use the same randomness) which renders decoding impossible.

$\mathsf{ChannelExp}_{\mathcal{A},\mathcal{LTS}_*}(\pi)$:

1. $\psi \leftarrow \perp$ ▷ Initial state of $\mathcal{A}$
2. $m_{\mathcal{R}} \leftarrow \perp$ ▷ Storage for decoded messages
3. $i = 1$ ▷ Number of queries
4. $s \leftarrow \mathsf{Gen}(1^\lambda)$ ▷ Secret-key generation
5. **while** $\mathcal{A}$ has a new query **do** ▷ Learning phase
   (a) $\mathcal{A}(\pi, \psi, m_{\mathcal{R}}) \rightarrow (\psi', m_i)$ ▷ Message selection
   (b) Generate a fresh nonce ℓ
   (c) Set $\pi' = (1^\lambda, s, \ell, \delta, \mathcal{D}, \varepsilon)$
   (d) Initialize oracle $\mathcal{O}_{m_i}$ to provide access to output symbols of $\mathsf{Encode}(\pi', m_i)$
   (e) $\mathcal{A}(\pi, \psi', m_i)^{\mathcal{O}_{m_i}(\cdot)} \rightarrow (\psi'', c_i)$ ▷ Codeword corruption where $c_i = (\sigma_1, \ldots, \sigma_{N_i})$
   (f) $\mathsf{Decode}(\pi', c_i) \rightarrow r_i$ ▷ Message recovery
   (g) Set $m_{\mathcal{R}} \leftarrow r_i$, $\psi \leftarrow \psi''$, $i \leftarrow i + 1$, and continue
6. **end while**
7. $\mathcal{A}(\pi, \psi, m_{\mathcal{R}}) \rightarrow (\psi', m_a)$ ▷ Attack phase
8. Generate a fresh nonce ℓ
9. Set $\pi' = (1^\lambda, s, \ell, \delta, \mathcal{D}, \varepsilon)$
10. Initialize $\mathcal{O}_{m_a}$ to access symbols of $\mathsf{Encode}(\pi', m_a)$
11. $\mathcal{A}(\pi, \psi', m_a)^{\mathcal{O}_{m_a}(\cdot)} \rightarrow (\psi'', c_a)$
12. $\mathsf{Decode}(\pi', c_a) \rightarrow r_a$
13. If $r_a \neq m_a$ and $r_a \neq \perp$ then output 1
14. Else if $r_a = \perp$ and $|c_a \cap Q_{m_a}| \geq (1 + \varepsilon)k$ then output 1
15. Else output 0

Figure 4.2: Security game for private LT-coding schemes.

Computationally-secure LT-coding Schemes. In normal operation over the random erasure channel, the probability that Decode fails is bounded by δ. We want to ensure that an adversary can neither: (1) cause Decode to output an incorrect message; nor (2) cause Decode to fail with probability significantly greater than δ.

We do this by defining *computationally secure (rateless or block) LT-coding schemes* where we wish to ensure that the adversary $\mathcal{A}$ is only negligibly more likely to cause a decoding error or failure than an adversary who attacks the codeword with random erasures.[24] Thus, we first define a *random adversary* $\mathcal{R}$ interacting in a restricted manner with $\mathcal{LTS}_*$ in the above game. $\mathcal{R}$ takes as input the same tuple of parameters π and a probability p, where $1 - p$ is non-negligible. Then $\mathcal{R}$ proceeds as follows: (1) it directly chooses an attack message $m_a \in \Sigma^k$ and outputs $(\perp, m_a)$, so that oracle $\mathcal{O}_{m_a}$ is initialized; (2) it queries $\mathcal{O}_{m_a}$ sequentially for encoded symbols; (3) for each retrieved symbol, it erases the symbol with probability p; otherwise it adds the symbol to a list (if the code has block-size n, then $\mathcal{R}$ erases at most pn symbols); (4) when $\mathcal{R}$ has more than $(1 + \varepsilon)k$ symbols (or

[24] We define security relative to a random channel rather than in absolute terms since the LT code itself reduces a random channel to a noiseless channel. That is, the cryptographic enhancements reduce the adversary to a random channel which is further reduced by the LT code to a noiseless channel.

at least $(1-p)n$ symbols) in the list, it outputs the list and exits. Note that $\mathcal{R}$'s output is distributed identically to REC_p.

Let $\mathcal{LTS}_*$ be a private (rateless or block) LT-coding scheme, and let $\mathsf{ExpAdv}_{\mathcal{A},\mathcal{LTS}_*}(\pi)$ be the experiment $\mathsf{ChannelExp}_{\mathcal{A},\mathcal{LTS}_*}(\pi)$ as defined above, where $\mathcal{A}$ is a PPT admissible adversary. Also, let $\mathsf{ExpRand}_{\mathcal{LTS}_*}(\pi,p)$ be $\mathsf{ChannelExp}_{\mathcal{R}_p,\mathcal{LTS}_*}(\pi)$ where $\mathcal{R}_p = \mathcal{R}(\pi,p)$ and p is (feasible) the erasure probability. Let $\mathsf{Adv}_{\mathcal{A},\mathcal{LTS}_*}(\pi,p) = |P[\mathsf{ExpAdv}_{\mathcal{A},\mathcal{LTS}_*}(\pi)=1] - P[\mathsf{ExpRand}_{\mathcal{LTS}_*}(\pi,p)=1]|$.

Definition 4.4. We say that a private (rateless or block) LT-coding scheme $\mathcal{LTS}_*$ is *computationally secure* (or just *secure*) if, for all PPT admissible adversaries $\mathcal{A}$ where $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$, and for all feasible $p \in (0,1)$ (or $p \in (0, 1-k/n)$ for block variants), we have that $\mathsf{Adv}_{\mathcal{A},\mathcal{LTS}_*}(\pi,p)$ is negligible in λ.

4.5 Core Falcon Codes

We now present our core technical constructions of three private LT-coding schemes. Based on LT codes and designed with a variety of efficiency goals, our new schemes define a broad class of *rateless (fountain) codes* as well as some corresponding *block-oriented refinements* which we generically term as *Falcon codes*. By construction, Falcon codes enjoy two desirable properties: they are designed to maintain the asymptotic performance of LT codes, thus allowing great degrees of efficiency, including fast encoding/decoding; and at the same time, they are crypto-enhanced to achieve strong error-correction capabilities. Our three LT-coding schemes described below comprise of: a *main scheme* Falcon that is rateless and particularly simple, a *scalable scheme* FalconS that is block refinement of our main scheme achieving better scalability, and a *randomized scheme* FalconR that is a rateless refinement of our scalable scheme. We refer to these as *core Falcon codes*, which in essence provide a new design framework for secure LT-based codes tolerant to malicious errors.

Our schemes are designed and analyzed for *static* data and fall to an adversary that can see updates to the data. Crucially, the schemes' security relies on hiding the structure of the underlying bipartite LT-coding graph. Any data updates would reveal the structure of this graph since a change to any input symbol would propagate to all associated code symbols. To prevent this information leakage, we would need to employ techniques such as Oblivious RAM (see, for instance, [49] and [154]). But, such techniques can be costly and need to "touch" much of the data when performing updates. We leave it as an open problem to construct efficient private LT codes for dynamic data.

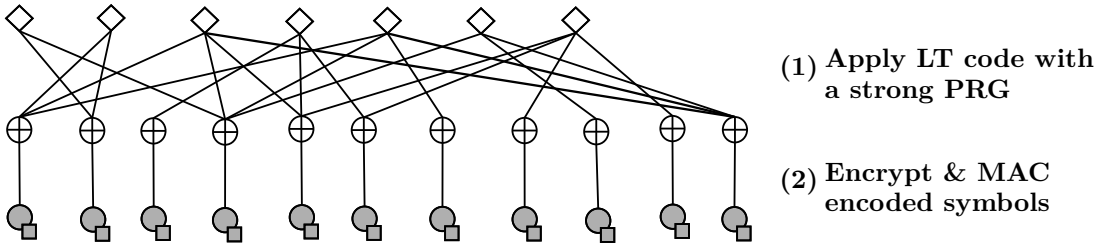


Figure 4.3: An example encoding via authenticated LT codes.

4.5.1 Main LT-coding Scheme

As explained in Section 4.3, LT codes [103] are a family of erasure codes of high efficiency, both theoretically (encoding/decoding time of $O(k \log k)$ for message length k) and practically. The encoder works by selecting (through the robust soliton distribution) $O(\log k)$ message symbols on average to combine to form a code symbol. The analysis in [103] was originally over the binary erasure channel, though it easily generalizes to larger symbols.

While there has been some work on extending LT codes to withstand errors (e.g., [104, 120, 129]), the studied channels have particular noise characteristics—such as additive white Gaussian or uniformly distributed noise—and adversarial errors are not considered. For Raptor codes, which are derived from LT codes, IETF standard RFC5053 [101] and its amendment RFC6330 [102], suggest using a simple checksum (e.g., CRC32) to detect any random corruptions of the encoded symbols. While this method may be sufficient for small, random errors, it will crumble quickly when faced with a malicious attacks.

Authenticated LT Codes. Our main scheme Falcon solves the challenge above by combining three cryptographic ingredients that are applied in a very simple and rather intuitive manner during LT-encoding (cf. Figure 4.3): a cryptographically strong PRG, a semantically secure cipher, and an existentially unforgeable MAC. (A nonce is also used to prevent replays.) First, the key change we perform in Encode compared to standard LT-encoding is to select each parity node's degree and its neighbors using the secure PRG. Then, after LT-coding, we encrypt all of the encoded (parity) symbols and compute a MAC tag for each encrypted symbol and the nonce (alternatively, we could use authenticated encryption with the nonce as additional authenticated data).[25] See also Algorithm 4.1.

[25] We also include each symbol's index i in the stream of symbols to prevent reorder and deletion attacks. When transmitting over a FIFO channel where, in addition, the receiver learns which symbols were erased, then we can omit

Algorithm 4.1 Encoder of main LT-coding scheme Falcon.

Input: 1^λ, keys k_{enc} and k_{mac}, master seed s, nonce ℓ, message size k, decoding failure probability δ, degree distribution $\mathcal{D}$, overhead ε, message m

Output: Authenticated codeword c

1: Set $s' \leftarrow f(s, \ell)$ ▷ f is a key-derivation function; use s' to seed the PRG
2: Set $i \leftarrow 0$ and $\pi \leftarrow (1^\lambda, s', k, \delta, \mathcal{D}, \varepsilon, m)$
3: Initialize LT-Encode(π)
4: **for** as long as required **do**
5: Set $\sigma \leftarrow$ LT-Encode(π, i) ▷ Generate the i-th LT code symbol σ
6: Set $e_i \leftarrow \mathsf{Enc}(k_{enc}, \sigma \circ i)$ ▷ Encrypt and MAC each symbol
7: Set $\tau_i \leftarrow \mathsf{Mac}(k_{mac}, e_i \circ \ell)$
8: Output $c_i \leftarrow e_i \circ \ell \circ \tau_i$
9: $i \leftarrow i + 1$
10: **end for**

Intuitively, the encryption, MACs, and secure PRG work together to maintain the "goodput" of the channel by ensuring that each (uncorrupted) received symbol is just as "helpful" for decoding as when sent over a random erasure channel.

Note, in Algorithm 4.1, we use the subroutine LT-Encode which takes as input the seed s for the PRG, message size k, decoding failure probability δ, degree distribution D, overhead ε, the message m, and an index i, and then outputs the i-th code symbol. Note that we also use a key-derivation function (KDF) f to generate a seed for the PRG using the master seed s and the nonce ℓ. Any any secure KDF can be used as long as the output is sufficiently long to seed the PRG. If we have message size k, decoding failure probability δ, degree distribution $\mathcal{D}$, and overhead ε, then we denote this as $(k, \delta, \mathcal{D}, \varepsilon)$-Falcon. The decoder for Falcon is shown in Algorithm 4.2 and is essentially the reverse of the encoder.

This simple design provides a secure LT code (see Section 4.6). First observe that the use of MACs ensures that any corruption in an encoded symbol is detected so that the symbol may be discarded. This step implements the standard known technique for reducing errors to erasures, that of authenticating the encoded symbols; we thus, informally, refer to our main scheme and its variants as an *authenticated LT code*.

But although a necessary condition, symbol verification is not sufficient for achieving security: input and parity symbols are interrelated through the underlying bipartite graph, so corruption of certain parity symbols may seriously disrupt recovery of certain input symbols. An adversary can partially infer this graph structure by looking at symbol *contents* (since code symbols are simply the

a symbol's index from the encoding.

Algorithm 4.2 Decoder of main LT-coding scheme Falcon.

Input: 1^λ, keys k_{enc}, k_{mac}, master seed s, nonce ℓ, message size k, decoding failure probability δ, degree distribution $\mathcal{D}$, overhead ε, authenticated codeword $c = (c_1, \ldots, c_N)$

Output: Message $m = (m_1, \ldots, m_k)$

Set $Rem = \emptyset$ (i.e., the empty set)
for $1 \leq j \leq N$ **do** ▷ Sieve out corrupted symbols
    Parse $c_j = e_j \circ \ell' \circ \tau_j$
    if $\ell' \neq \ell$ or $\mathsf{VerifyMac}(k_{mac}, e_j \circ \ell, \tau_j) = 0$ **then**
        Discard c_j and continue
    else
        $\sigma_j \circ j' \leftarrow \mathsf{Dec}(k_{enc}, e_j)$
        $Rem \leftarrow Rem \cup \{(\sigma_j, j')\}$
    end if
end for
Set $s' \leftarrow f(s, \ell)$ ▷ f is a key-derivation function; use s' to seed the PRG
Output $m \leftarrow \mathsf{LT\text{-}Decode}(1^\lambda, s', k, \delta, \mathcal{D}, \varepsilon, Rem)$

XOR of a random subset of message symbols) and target specific symbols for erasure, seeking to maximally disrupt decoding.[26] Symbol encryption ensures that the content of each symbol (and any information about the graph structure contained therein) is hidden from the adversary. Similarly, the strong PRG ensures that the adversary cannot exploit any biases or weaknesses in the PRG (e.g., output that is initially biased, cf. RC4) to aid in guessing what the digraph structure may be. Overall, as the structure of the graph is unpredictable, an adversary will, intuitively, be unable to do better than random corruptions and erasures.

As a side effect of our use of encryption, Falcon also provides message privacy. In situations where the encoded data is encrypted by the channel (e.g., tunneling over SSH), then Falcon need only use a PRG and a MAC, reducing the overhead of our modifications. If Falcon is used in combination with an encrypted and authenticated channel—e.g., IPsec—then only a PRG is needed and the overhead is further reduced. (In the case of IPsec, however, "PRG-only" Falcon would be vulnerable to malicious proxies.)

Batching. Note that secure MACs require input lengths to be at least as large as the security parameter, thus Falcon operates on message alphabets where each symbol is at least λ bits in size, i.e., typically at least 10 or 16 bytes. Authenticating multiple symbols with a single MAC (as a single *batch*) is a reasonable implementation in many circumstances—for instance, when grouping symbols together to be sent in a single network packet. Note, however, that this results in corruption amplification

[26] E.g., for a message containing a single 1 bit (the rest being 0's), $\mathcal{A}$ can easily discern which parity symbols include the non-zero message symbol.

during decoding (a single symbol corruption can cause many other possibly valid symbols to be discarded). But, in channels where errors are of low rate or are *bursty*, authenticating a batch of symbols would increase throughput since the cryptographic overhead is reduced. (Additionally, to further reduce overhead, we note that a batch can be treated as a single plaintext and encrypted as a unit.) Batching also loosens the lower-bound on the symbol size and instead requires that the batch is at least λ bits. Note that batching increases the decoding overhead ε since $(1+\varepsilon)k$ is not necessarily divisible by the batch size. With a batch of b symbols, the decoder must receive $(1+\varepsilon)k \leq \lceil(1+\varepsilon)k/b\rceil b \leq (1+\varepsilon)k + b - 1$ symbols, giving an effective overhead of $\varepsilon \leq \varepsilon' \leq \varepsilon + \frac{b-1}{k}$. Since the presence or absence of batching does not affect our security analyses (except the fixed-rate block-oriented code, detailed next), we assume batches of size 1.

Finally, we note that appending MACs to parity symbols increases the space overhead per symbol. For both block and rateless codes, the number of raw bits transmitted increases by a factor of $(1 + m/s)$, where m is the MAC size and s is the symbol size. For instance, if $s = m$ (the minimum symbol size), then size of an output symbol is doubled. However, if $m \ll s$, then this overhead is small (e.g., for $s = 1024$ bytes and $m = 16$ bytes, the overhead is $\approx 1.5\%$). Batching b symbols per MAC decreases the overhead to $m/(bs + m)$.

4.5.2 Scalable (Block) LT-coding Scheme

Although simple and efficient, the authenticated LT code presented above suffers from scalability problems in certain cases. In particular, if the input file and the internal state of the encoder do not fit into main memory, then the operating system will need to continually swap pages in and out. Then, on average, $O(\log k)$ random disk reads will be needed to produce a single parity symbol, as any such symbol depends on randomly selected message symbols. Magnetic disks have random read latencies around 5–10 milliseconds while SSDs have read latencies on the order of 10s of microseconds. But both of these are orders of magnitude larger than the 10s of nanoseconds required to read from RAM. Hence, the large amount of IO required when paging in and out will drastically slowdown the encoder and decoder (see Figure 4.13 in Section 4.7).

We can mitigate this limitation by adopting a simple divide-and-conquer strategy on our main scheme Falcon, towards the design of a new *scalable* scheme FalconS: we divide the input into blocks (of symbols) and then encode each block independently using Falcon. This immediately increases data locality during message encoding, which provides better scalability (as only a portion of the input

must reside in memory at any given time) and further allows for easy parallelization (see Figures 4.9, 4.10, and 4.14 in Section 4.7). This method, though, introduces a new security concern: depending on parameterization, an adversary may apply all of its effort to corrupt parity symbols in a single (a few selected) block(s), thus significantly increasing its advantage in causing a decoding failure, even if all block encoders have produced symbols beyond the LT-decoding threshold.

To defend against this attack type, we adopt a technique due to Lipton [98]. We apply a *random permutation* π over all produced parity symbols across all blocks. This ensures that any corruptions performed by an adversary are distributed uniformly both *among* all blocks and *within* each block which, in turn, allows Falcon to use a weaker and faster PRG when producing an individual block encoding, since any corruption on this block cannot be targeted, but only random. Applying π over all parity symbols, though, necessitates a *fixed upper bound* on their total number, thus making FalconS a *block* code—this can be considered a useful byproduct of our new scheme. But employing (rateless) Falcon within (block) FalconS, as above, requires careful encoding parameterization. Given an adversarial *corruption rate* γ—with which an adversary corrupts a γ-fraction of all symbols, or a γ-fraction of batches—and by applying a random permutation π to all of the code symbols from all of the blocks, we have that the number of symbol corruptions per block are *binomially distributed*. Therefore, we add extra redundancy to each encoded block produced by Falcon, expressed by *tolerance rate* τ, to absorb any variance in per-block errors—otherwise, block (and also total) decoding fails.[27]

Algorithm 4.3 details the above encoding using permutation π *explicitly*—we call this variant of our scalable (block) scheme FalconSe—but it is possible for the permutation to be applied implicitly, which we detail next. As before, we use a secure KDF f to derive a session seed for G using the master seed s and nonce ℓ. The algorithm as shown does not simply use Falcon as a black-box subroutine; rather, the Falcon encoding algorithm directly integrated. This is done to include the block index i with each message symbol which is used to ensure that any re-ordering of symbols by the adversarial channel can be mitigated.[28] The decoder for FalconSe is shown in Algorithm 4.4 and is essentially the reverse of the encoder.

However, there is a drawback to using an explicit permutation π: new parity symbols must be buffered until they are all generated (and then can be permuted!).[29] If resources are constrained, this

[27] These losses can be mitigated by applying an additional level of erasure coding to the input before breaking it up into blocks, but any erasure code can only tolerate a certain number of block losses. In any case, we must bound the probability that a block receives "too many" corrupted symbols.

[28] This is necessary even if the channel is FIFO, since we also allow the adversary to delete symbols.

[29] At most, symbols can be output piece-meal by pre-computing π and placing a generated symbol in its final place.

Algorithm 4.3 Encoder of scalable LT-coding scheme FalconSe.

Input: 1^λ, keys k_{enc} and k_{mac}, master seed s, nonce ℓ, symbols per block k, block decoding failure probability δ, degree distribution $\mathcal{D}$, overhead per block ε, corruption rate γ, block-corruption tolerance τ, number of blocks b, message m

Output: Authenticated codeword c

Set $s' \leftarrow f(s, \ell)$ ▷ f is a key-derivation function
Seed PRG G with s'
Set $M \leftarrow \frac{1}{1-(1+\tau)\gamma}(1+\varepsilon)k$
Generate random permutation Π of $b * M$ elements
Partition m into blocks $B_1, \ldots, B_b$
for $1 \leq i \leq b$ **do**
    Generate a seed s_i using G
    Set $\pi \leftarrow (1^\lambda, s_i, k, \delta, \mathcal{D}, \varepsilon, M)$
    Use a weak (but fast) PRG H as the source of randomness for LT-Encode
    Set $(\sigma_{i,1}, \ldots, \sigma_{i,M}) \leftarrow$ LT-Encode(π, B_i) ▷ Compute M parity symbols over B_i
    for $1 \leq j \leq M$ **do**
        Set $e_{i,j} \leftarrow$ Enc$(k_{enc}, \sigma_{i,j} \circ i \circ j)$
        Set $\tau_{i,j} \leftarrow$ Mac$(k_{mac}, e_{i,j} \circ \ell)$
        Set $c_{i,j} \leftarrow e_{i,j} \circ \ell \circ \tau_{i,j}$
    end for
end for
Map each $c_{i,j}$ to position $\Pi(i * M + j)$ to get $c' = (c'_1, \ldots, c'_{bM})$
Output $c' = (c'_1, \ldots, c'_{bM})$

extra buffering can demand too much memory, induce swapping, and greatly reduce performance. In such a situation an *implicit* permutation would be better (see Figure 4.13 in Section 4.7). In this variant, π is generated *first* and then parity symbols are produced (by Falcon from the appropriate block encoder) *in the order of their final position* (i.e., after π would have been applied). This allows a Falcon encoder to output parity symbols in a streaming manner (but out-of-order). We call this variant of our scalable scheme FalconSi. For FalconSi or FalconSe, if we have k message symbols per-block, block decoding failure probability δ, degree distribution $\mathcal{D}$, overhead ε, an adversarial corruption rate of γ, corruption-tolerance parameter τ, and b blocks in the encoding, we denote this as $(k, \delta, \mathcal{D}, \varepsilon, \gamma, \tau, b)$-FalconS, where FalconS can be either FalconSe or FalconSi.

To minimize the overhead of managing the implicit permutation, it is desirable that the PRG state in each encoder—used to generate the bipartite graph—can be easily and quickly reset to produce a desired segment of pseudorandomness (like in the Salsa20 stream cipher [15]). That is, the PRG state would be reset to produce the pseudorandom bits that would have been output if the symbol generation was done in order.[30] If this feature is not present, all needed pseudorandom bits

But the output could be blocked for some time waiting for the appropriate symbol to be produced.

[30] This would, of course, require the pre-computation of all node degrees; otherwise, it is impossible to know "where" a given node's randomness lies.

Algorithm 4.4 Decoder of scalable LT-coding scheme FalconSe.

Input: 1^λ, keys k_{enc}, k_{mac}, master seed s, nonce ℓ, symbols per block k, block decoding failure probability δ, degree distribution $\mathcal{D}$, overhead per block ε, corruption rate γ, block corruption-tolerance τ, number of blocks b, authenticated codeword $c = (c_1, \ldots, c_{bM})$

Output: message m or $\perp$

Set $M \leftarrow \frac{1}{1-(1+\tau)\gamma}(1+\varepsilon)k$
Set $s' \leftarrow f(s, \ell)$ ▷ f is a key-derivation function
Seed the PRG G using s
Generate a random permutation Π over bM elements
Set $B_1 \leftarrow \emptyset, \ldots, B_b \leftarrow \emptyset$
for $1 \le i \le bM$ **do** ▷ De-permute and gather into blocks
    Parse $c_i = (e_i, \ell', \tau_i)$
    if $\ell' \neq \ell$ or $\mathsf{VerifyMac}(k_{mac}, e_i \circ \ell, \tau_i) = 0$ **then**
        Discard c_i and continue
    end if
    Set $\sigma_i \circ i' \circ j' \leftarrow \mathsf{Dec}(k_{enc}, e_i)$
    Set $B_{i'} \leftarrow B_{i'} \cup \{(\sigma_{j'}, j')\}$
end for
for $1 \le j \le b$ **do**
    Generate a seed s_j using G
    Use (weak) PRG H as the source of randomness for LT-Decode
    Set $\pi \leftarrow (1^\lambda, s_j, k, \delta, \mathcal{D}, \varepsilon)$
    Set $m_j = (m_{j,1}, \ldots, m_{j,k}) \leftarrow \mathsf{LT\text{-}Decode}(\pi, B_j)$
    if $m_j = \perp$ **then** ▷ If decoding fails
        Output $\perp$ and exit
    end if
end for
Output $m = (m_{1,1}, \ldots, m_{b,k})$

can be pre-computed (namely, computing the degree and neighbors of a parity node). Fortunately, the space needed to store the pre-computed bits can be much less than what would be required to store the output symbols, e.g., in storage applications with large message symbols (say 1KB in size). For example, say we have $k = 2^{16}$ input symbols and that the average node degree is $\log k = 16$. Then, a given node will only require, on average, $2 * 16 + 2 = 34$ bytes to store the degree and neighbor indices (we assume that the degree can fit in 2 bytes).

Decoding Failure Probability. Suppose we want to achieve an overall decoding failure probability of δ for this scheme. Decoding fails when any of the individual blocks fail to decode, and this could be from either too many corrupted symbols in a block, so that there are fewer than $(1+\varepsilon)k$ good symbols, or we were unlucky and the code symbols did not cover all input symbols. The probability of the first case can be made negligible via the corruption tolerance parameter τ; the second case is intrinsic to the schemes themselves. Note that in this case, the failures are *independent* and so are the

successes. Hence, the probability of decoding success is $(1-\delta')^b = 1-\delta$ and solving for δ', we have $\delta' = 1 - \sqrt[b]{1-\delta}$. For example, with $b = 100$ and $\delta = 0.05$, we have $\delta' = 1 - \sqrt[100]{1-0.05} \approx 0.000512$.[31] Note that the smaller value of δ' implies that the decoding overhead for each block ε' is increased.

Error Analysis. In FalconS, the symbols of each block are all combined together uniformly at random, giving us, via a balls-in-bins analysis, that the number of corruptions in a given block is binomially distributed. Hence, there is no way to ensure that each block receives at most a fixed number of corruptions. To mitigate this, we add additional redundancy to each block to absorb the variance in the number of corrupted symbols per block, increasing the total number of code symbols generated by a factor of $(1+\tau)$, where τ is the *tolerance rate*. Ideally, we wish to minimize this extra redundancy while also ensuring that the probability that we exceed the error-correction capacity of a block is negligible in λ.

Suppose, there are b blocks, where each block has k input symbols and is encoded into m output symbols, giving $n = bm$ total symbols. Suppose a γ-fraction of symbols are corrupted. If we encode a block so that it can tolerate $(1+\tau)\gamma m$ corruptions (were γm is the mean number of corruptions per block), then we must generate $m = \frac{1}{1-(1+\tau)\gamma}(1+\varepsilon)k$ symbols per block. We use a Chernoff-bound to bound the probability that there are more than $(1+\tau)\gamma$ corruptions in a given block. For a binomially distributed random variable X with mean μ and for some $\tau > 0$ it holds that,

$$P(X \geq (1+\tau)\mu) \leq \left(\frac{e^\tau}{(1+\tau)^{(1+\tau)}}\right)^\mu.$$

For our parameters, we have that $\mu = \gamma\frac{1}{b}n = \gamma m$. Suppose we want the probability to be less than some value q. Note that, $(\frac{e^\tau}{(1+\tau)^{(1+\tau)}})^\mu \leq q$ is not solvable algebraically in terms of τ. However, it can be solved numerically. Consider the following parameterization where the message consists of $k = 15000$ symbols in each of $b = 100$ blocks and the decoding overhead is $\varepsilon = 0.05$. The adversary $\mathcal{A}$ corrupts a $\gamma = 0.2$-fraction of the output symbols and we add a τ-fraction additional redundancy to each block. Thus, each block consists of $m = \frac{1}{1-0.2(1+\tau)}(1+0.05)15000 = \frac{15750}{0.8-0.2\tau}$ symbols and the average number of corrupted symbols is $\gamma m = \frac{3150}{0.8-0.2\tau}$. Suppose that we want $P(X \geq (1+\tau)\mu) \leq q = 2^{-128}$. Solving for τ, we have that $\tau \approx 0.21354$. If we calculate the tail of the distribution exactly, we find that the value of τ is close to 0.20788. So, the approximation overestimated the necessary redundancy by just 2.7%, and for larger values of q the overestimation is smaller.

[31] If we apply a $[n, b]$ erasure code across the blocks, then the probability of decoding success (i.e., that at most $n-b$ blocks fail to decode) is $\sum_{i=0}^{n-b}\binom{n}{i}(1-\delta')^{n-i}(\delta')^i$, which can be solved for δ' via numerical methods.

Asymptotic Efficiency. We wish for our FalconS codes to achieve the $O(k \log k)$ encoding/decoding time for each block that LT codes (and our Falcon codes) achieve. The permutation step can be performed in linear time by using the Fisher-Yates algorithm [42], but the extra redundancy added requires a careful analysis. We show next that the tolerance parameter τ is $o(1)$, and so we maintain $O(k \log k)$ encoding and decoding. Recall the Chernoff-bound above, where $\mu = \gamma m = \frac{\gamma}{1-(1+\tau)\gamma}(1+\varepsilon)k$ is the mean number of corruptions per block. Let $\mu' = \frac{\gamma}{1-\gamma}(1+\varepsilon)k$, then

$$\left(\frac{e^{\tau}}{(1+\tau)^{(1+\tau)}}\right)^{\mu} \leq \left(\frac{e^{\tau}}{(1+\tau)^{(1+\tau)}}\right)^{\mu'}.$$

If we bound the right-hand side by q, then we have,

$$\mu' \ln\left(\frac{e^{\tau}}{(1+\tau)^{(1+\tau)}}\right) \leq \ln q$$

$$\mu'(\tau - (1+\tau)\ln(1+\tau)) \leq \ln q$$

$$(1+\tau)\ln(1+\tau) - \tau \geq \frac{-\ln q}{\mu'}.$$

Since the left-hand side is monotonically increasing, to minimize τ we must set the two sides equal. Thus we have,

$$(1+\tau)\ln(1+\tau) - \tau = \frac{-\ln q}{\mu'} = \frac{-(1-\gamma)\ln q}{\gamma(1+\varepsilon)k}.$$

Since γ, q, and ε are constants, we have $(1+\tau)\ln(1+\tau) - \tau = O(\frac{1}{k}) = o(1)$. Since $\tau = o((1+\tau)\ln(1+\tau))$, this implies $\tau = o(1)$; i.e., the amount of extra redundancy is bounded by a constant and we preserve $O(k \log k)$ encoding and decoding times.

4.5.3 Randomized Scalable LT-coding Scheme

Our block-oriented code FalconS does indeed achieve better scalability than Falcon. (As we show in Section 4.7, it can, for instance, provide over a 50% speed-up for an input file 32MB in size.) However, FalconS codes are inherently no longer rateless codes: any permutation that is explicitly/implicitly applied to all symbols across all blocks precludes the possibility of rateless encoding.[32]

We now present an alternative approach that is block-oriented, thus still scalable, yet allows for rateless encoding. As before, the idea is to break the input up into blocks, and then apply a Falcon code to each block. However, the key idea now is to produce symbols for the output encoding

[32] To create the permutation, the number of code symbols must be known.

Algorithm 4.5 Encoder of FalconR scalable LT-coding scheme.

Input: 1^λ, keys k_{enc}, k_{mac}, master seed s, nonce ℓ, symbols per block k, block decoding failure probability δ, degree distribution $\mathcal{D}$, overhead per block ε, number of blocks b, message m
Output: authenticated codeword c

Set $s' \leftarrow f(s, \ell)$ ▷ f is a key-derivation function
Seed G with s'
Generate b seeds $s_1, \ldots, s_b$ using G
Divide m into b blocks $m_1, \ldots, m_b$
Set $\pi_i \leftarrow (s_i, k, \delta, \mathcal{D}, \varepsilon)$ for $1 \leq i \leq b$
Initialize b encoders $\mathsf{LT\text{-}Encode}_1(\pi_1, m_1), \ldots, \mathsf{LT\text{-}Encode}_b(\pi_b, m_b)$
for as long as required **do**
    Sample a block index i from G
    Let σ be the j-th symbol of $\mathsf{LT\text{-}Encode}_i$
    Let $e \leftarrow \mathsf{Enc}(k_{enc}, \sigma \circ i \circ j)$
    Let $\tau \leftarrow \mathsf{Mac}(k_{mac}, e \circ \ell)$
    Output $e \circ \ell \circ \tau$
end for

by applying Falcon codes *in parallel* to the blocks, giving the randomized, scalable, and rateless scheme FalconR. In particular, for each block an independent instance of Falcon is initialized with a unique random seed generated by a strong PRG. Then, another strong PRG, keyed with another random seed, is used to iteratively select a block *at random* whose encoder will simply output its next symbol; this random selection is repeated (at least) until *each* block encoding has reached the required decoding threshold. See Algorithm 4.5 for the encoder and Algorithm 4.6 for the decoder. As before, we use a KDF f to derive a seed for G using the master seed s and nonce ℓ. As with FalconS, we integrate Falcon directly into the FalconR encoder, allowing us to include with each symbol the index of its block, mitigating symbol deletion and reordering attacks. If we have b blocks and encode each using a $(k, \delta, \mathcal{D}, \varepsilon)$-Falcon code, then we denote this $(k, \delta, \mathcal{D}, \varepsilon, b)$-FalconR.

We emphasize the subtle difference between codes FalconSi and FalconR: in a FalconSi code, the encoder for each block produces its symbols in a random order *induced by* π, but in FalconR code these are produced in the (correct) order *induced by the block encoder itself.*

FalconR codes retain the rateless property of the original Falcon codes: new symbols can be produced by continuing to select blocks at random and outputting symbols. The security of FalconR reduces to the security of Falcon codes: the random selection of encoders ensures that adversarial corruptions are randomly and uniformly distributed among the blocks preventing too many corruptions from landing in any one block. Moreover, the secure encoder used on each block ensures that, for any corruptions that occur in that block, the adversary can do no better than a random channel. Note

Algorithm 4.6 Decoder of FalconR scalable LT-coding scheme.

Input: 1^λ, keys k_{enc} and k_{mac}, master seed s, nonce ℓ, symbols per block k, block decoding failure probability δ, degree distribution $\mathcal{D}$, overhead ε per block, number of blocks b, authenticated codeword c

Output: message m or $\perp$

Set $s' \leftarrow f(s, \ell)$ ▷ f is a key-derivation function
Seed the PRG G using s'
Generate b seeds $s_1, \ldots, s_b$ using G ▷ Used for the strong PRGs in the LT-encoders
Set $\pi_i \leftarrow (1^\lambda, s_i, k, \delta, \mathcal{D}, \varepsilon)$ for $1 \le i \le b$
Initialize b decoders $\mathsf{LT\text{-}Decode}_1(\pi_1), \ldots, \mathsf{LT\text{-}Decode}_b(\pi_b)$ ▷ $\mathsf{LT\text{-}Decode}_i$ decodes the i-th block
for each symbol c_i in c **do**
    Parse $c_i = (e_i, \ell', \tau_i)$
    if $\ell \neq \ell'$ or $\mathsf{VerifyMac}(k_{mac}, e_i \circ \ell, \tau_i) = 0$ **then**
        Discard c_i and continue
    end if
    Set $\sigma_i \circ i' \circ j' \leftarrow \mathsf{Dec}(k_{enc}, e_i)$
    Update $\mathsf{LT\text{-}Decode}_{i'}$ with σ_i as its j'-th symbol
end for
for $1 \le i \le b$ **do**
    Let $(m_{i,1}, \ldots, m_{i,k})$ be the message symbols output by $\mathsf{LT\text{-}Decode}_i$
    If $(m_{i,1}, \ldots, m_{i,k}) = (\perp, \ldots, \perp)$, then output $\perp$ and exit
    ▷ If any block fails to decode, output nothing
end for
Output $m = (m_{1,1}, \ldots, m_{b,k})$

that while FalconS can use a weak PRG when encoding an individual block, here each instance of Falcon must use a strong PRG.

Efficiency. The asymptotic efficiency of our randomized scheme is not immediately obvious. Enough encoded symbols must be produced for each block, namely at least m code symbols, for some $m = \Omega(k)$, where k is the number of input symbols per block. This problem is a generalization of the standard balls-in-bins problem which asks how many balls must be thrown at random into b bins to ensure that each bin has at least one ball. Here, what is relevant is how many balls must be thrown into b bins to get at least m balls in each bin. In [117], it is shown that, as b goes to infinity, the expected number of balls thrown is $b[\log b + (m-1)\log\log b + C_m + o(1)]$, for some constant C_m. That is, the expected number is $b\log b + b(m-1)\log\log b + O(b)$. Thus, on average, we gain (at least) an additional $\log\log b$ factor in the time to encode and decode the file.[33] The tolerance parameter τ of FalconS is no longer needed here; each block uses the Falcon encoder and FalconR is rateless, so more symbols can always be produced.

[33] If we use batching, then the same analysis applies if we have the "transmission unit" be a batch instead of a single symbol.

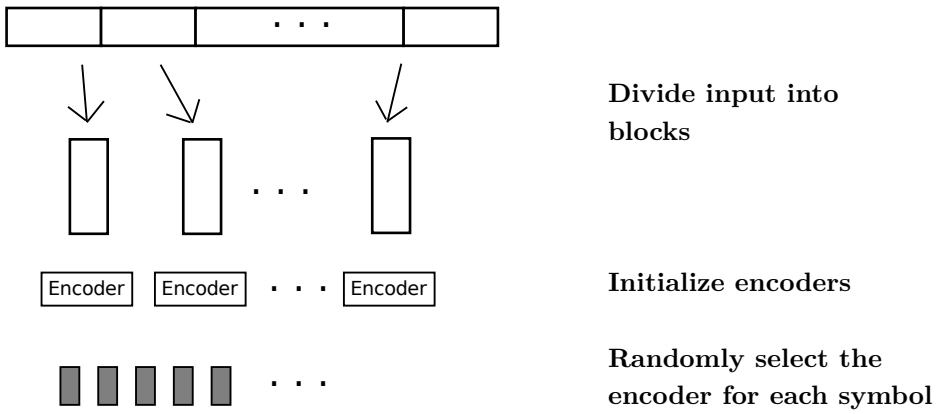


Figure 4.4: Diagram illustrating the FalconR encoder.

If, however, we have a small b (e.g., $b = 10$), then since k must be sufficiently large (and hence, so must m), by the law of large numbers the average number of balls thrown is $O(bm)$. The asymptotic behavior of FalconR as b grows is important since we would like this scheme to scale up to very large files (e.g., 10s of gigabytes or more). Suppose the degree distribution requires 12000 code symbols to recover the input with high probability, and that symbols are at least 10 bytes in size, then we get an implicit lower-bound on block size of about 128KB (less if symbols in a block are batched together). As a 1GB file will contain approximately 8000 such blocks, both the case where there are many blocks and the case where there are few must be considered.

4.6 Security Analyses

Our various constructions of Falcon codes seek to reduce an adversarial corrupting channel to a *random erasure channel* (REC). Through such a reduction, our LT-coding schemes inherit many of the properties of the original LT codes (e.g., the overhead ε and failure probability δ). We next provide proofs of the security of our core Falcon code schemes. For simplicity and clarity, the theorems and lemmas are stated using asymptotic security, but several of our proofs are exact where the resources used by the adversary are specified.

The adversary $\mathcal{A}$ can win the game ChannelExp by causing a decoding failure (Decode outputs $\perp$) or a decoding error (Decode outputs an incorrect message). Since these are mutually exclusive events, they are considered separately in the following lemmas. Lemma 4.1 states that the ability

of $\mathcal{A}$ to cause a decoding error in authenticated LT code is directly reducible to the security of the message authentication code. Lemma 4.2 states that the probability of causing decoding failure is only negligibly different from a random channel. Recall that, as stated in Section 4.4, we only consider admissible adversaries $\mathcal{A}$ that are compared to random erasure channels with a feasible erasure probability p.

Intuitively, in Lemma 4.1, for $\mathcal{A}$ to force Decode to make a decoding error, it must accept at least one corrupt code symbol, which can only happen if the MAC for the symbol has been forged. Recall that $\mathcal{O}_m$ denotes the encoder Encode initialized to encode message m. A query of i to $\mathcal{O}_m$ returns the i-th code symbol. Recall that F is a relation that holds when given ε, $\mathcal{D}$, and δ, using distribution $\mathcal{D}$ in the LT-encoding, after receiving $(1+\varepsilon)k$ symbols decoding succeeds with probability at least $1-\delta$.

Lemma 4.1. *Let $M = (\mathsf{Gen}_1, \mathsf{Mac}, \mathsf{VerifyMac})$ be an existentially-unforgeable MAC used to authenticate the symbols of a $(k, \delta, \mathcal{D}, \varepsilon)$-Falcon code $\mathcal{LTS}$, where $F(\delta, \mathcal{D}, \varepsilon)$ holds, and let λ be the security parameter. For all sufficiently large k and for all PPT $\mathcal{A}$, the probability that $\mathcal{A}$ wins $\mathsf{ChannelExp}_{\mathcal{A},\mathcal{LTS}}$ via a decoding error is negligible in λ.*

Proof. First, suppose that the adversary $\mathcal{A}$ that causes a decoding error, with $\mathcal{A}$ running in time T, making q queries in the learning phase (each of which is comprised of at most n symbols), and succeeding with advantage ϵ. Since $\mathcal{A}$ seeks to cause a decoding *error*, it will not make corruptions in an attempt to cause a decoding *failure*. So the probability that decoding fails is still δ and the probability of a decoding error is ϵ. We will construct an algorithm $\mathcal{A}'$ that breaks M and runs in polynomial time, making at most $(q+1)n$ MAC oracle queries, and succeeds with probability at least $\frac{\epsilon}{n}$ (and at most ϵ).

$\mathcal{A}'$ is given access to a MAC oracle $\mathcal{O}_{mac}$ and simulates the ChannelExp game, using $\mathcal{A}$ as a subroutine. Given an $\mathcal{LTS}$ code with parameters $k, \varepsilon, \mathcal{D}, \delta$ and security parameter 1^λ, on input 1^λ $\mathcal{A}'$ runs the ChannelExp with parameters $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$. For $\mathcal{A}'$ to produce a successful forgery, we require that none of its queries to $\mathcal{O}_{mac}$ are used as the final message output by $\mathcal{A}'$.

To simulate $\mathsf{ChannelExp}(\pi)$, $\mathcal{A}'$ uses $\mathcal{O}_{mac}$ to create the MACs for the code symbols when simulating Encode. To simulate Decode, $\mathcal{A}'$ tracks the symbol-MACs pairs from the queries from $\mathcal{A}$ and uses this list to filter out the symbols output by $\mathcal{A}$ that were either not previously queried or have a MAC that is different from the result given by $\mathcal{O}_{mac}$ (i.e., the symbols that contain forged MACs). After q rounds in the learning phase, $\mathcal{A}$ outputs m_a for the attack phase. After simulating Encode and

giving the result to $\mathcal{A}$, the algorithm $\mathcal{A}$ outputs the corrupted code symbols $(\sigma_1, \ldots, \sigma_{n'})$ (for some $(1+\varepsilon)k \leq n' \leq n$).

We require that the output of $\mathcal{A}'$ was not queried to $\mathcal{O}_{mac}$ and so can exclude those symbol-MAC pairs that match the Encode oracle queries.[34] But, there are at most n bad symbols to choose from. $\mathcal{A}'$ selects one symbol-MAC pair from the remaining pairs at random and outputs it as the attempted forgery. Since $\mathcal{A}$ succeeds with advantage at most ϵ, $\mathcal{A}'$ succeeds with probability at least $\frac{\epsilon}{n}$. Thus, $\mathcal{A}'$ runs in time $T+O(qn)$, making at most $(q+1)n$ queries to $\mathcal{O}_{mac}$ (with T and q polynomially-bounded), and succeeds with advantage $\frac{\epsilon}{n} \leq \epsilon' \leq \epsilon$. Since M is existentially unforgeable, it must be that ϵ' is negligible; hence, ϵ is negligible. □

For the next lemma, we will prove that the advantage of any adversary in causing a decoding *failure* is only negligibly greater than δ. Roughly, the proof proceeds as follows. Since we are using a semantically secure cipher to encrypt the code symbols, the codeword c leaks no "information" about the underlying encoding of m to the adversary $\mathcal{A}$. Suppose $\mathcal{A}$ succeeds with probability μ. The semantic security of the cipher implies that there exists an $\mathcal{A}'$ that *without* access to c—that is, *independent* of c—but given some information about m, outputs a set of indices of symbols to be erased such that Decode fails with probability $\approx \mu$ (i.e., negligibly different from μ). Since the symbols to be erased are chosen independently of the encoding c, the remaining symbols form a random graph over the input symbols and, thus, $\mathcal{A}'$ succeeds with probability exactly δ. Thus, $\mathcal{A}$'s advantage (i.e., $|\mu - \delta|$) is at most negligible.

Lemma 4.2. *Let $\mathcal{LTS}$ be a $(k, \delta, \mathcal{D}, \varepsilon)$-$\mathsf{Falcon}$ code where the relation $F(\delta, \mathcal{D}, \varepsilon)$ holds, let $\Pi = (\mathsf{Gen}_2, \mathsf{Enc}, \mathsf{Dec})$ be a semantically secure symmetric cipher used by $\mathcal{LTS}$, and let G be a secure PRG used by $\mathcal{LTS}$. Then, for all sufficiently large k and for all PPT admissible $\mathcal{A}$, the probability that $\mathcal{A}$ wins $\mathsf{ChannelExp}_{\mathcal{A},\mathcal{LTS}}$ via a decoding failure is negligibly different from δ.*

Proof of Lemma 4.2. Suppose we have an adversary $\mathcal{A}$ that causes a decoding failure with probability μ. Initially, assume that $\mathcal{A}$ only erases symbols instead of corrupting them (we will remove this assumption later). Let $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$ be the parameters for a Falcon coding scheme $\mathcal{LTS}$. Define $h(m)$ (the "history" of m) to be $h(m) = (\pi, \psi', m)$.

Recall that $\mathcal{O}_m$ is an encoding oracle for $\mathcal{LTS}$ initialized to encode message m, where on input i, $\mathcal{O}_m$ outputs the i-th code symbol for the encoding of m. $\mathcal{O}_m$ is reinitialized with a new m chosen by $\mathcal{A}$

[34] Since M is existentially unforgeable, this is indistinguishable from having a MAC verification oracle.

at each round. $\mathcal{A}$ takes as input π, ψ', runs in time t, and is given access to $\mathcal{O}_m$. After interacting with $\mathcal{O}_m$, $\mathcal{A}$ outputs $(\psi'', \sigma'_1, \ldots, \sigma'_n)$, where $\sigma'_i \in \Sigma \cup \{\perp\}$ (Σ is the code alphabet and $\perp$ is an erasure).

Define $\mathcal{B}$, such that, on input (π, ψ', m) and access to $\mathcal{O}_m$, it outputs $(i_1, \ldots, i_{n'})$, where $n' \leq n - (1+\varepsilon)k$, n is the total number of symbols requested from $\mathcal{O}_m$ by $\mathcal{A}$, and each i_j is the index of an erased symbol. Note that we can construct $\mathcal{B}$ by simply using $\mathcal{A}$ as a subroutine and outputting the indices of all $\perp$ symbols. We define $\mathcal{B}$ to be *successful* if, after erasing the symbols corresponding to the i_j's in the encoding of m, and giving the result to Decode, decoding fails. Note that $\mathcal{B}$ succeeds exactly when $\mathcal{A}$ does and runs in time $t_{\mathcal{B}} = t + O(n)$.

Since we are using a semantically secure cipher Π, there exists $\mathcal{B}'_1$ such that on input $h(m)$ and *without* access to the first code symbol, $\mathcal{B}'$ outputs $(i_1, \ldots, i_{n'})$, where n' and each i_j are as before. Let γ denote the difference in the advantage of $\mathcal{B}'_1$ and $\mathcal{B}$. Note that $\mathcal{B}'_1$ runs in time $t_{\mathcal{B}} + o$ where o is the "overhead" in running time that $\mathcal{B}_1$ requires (o is polynomially bounded). Now, define $\mathcal{B}'_i$ such that $\mathcal{B}'_i$ does not have access to the first i symbols of the encoding of m. By induction, we have that $\mathcal{B}'_i$'s probability of success is at most $i\gamma$ different from $\mathcal{B}$ and its running time is at most $t_{\mathcal{B}} + io$.

Note that $\mathcal{B}'_n$ succeeds with probability at most $n\gamma$ different from μ without seeing the encoding of m at all. This implies that $\mathcal{B}'_n$ can be run and select which symbols to erase *independently* of the encoding of m. Hence, after applying the output of $\mathcal{B}'_n$ to c, the remaining unerased symbols form a *random graph* over the symbols of m, where the degree distribution for the uncorrupted symbols is identical to the distribution used to encode m. Thus we have that, given $(1+\varepsilon)k$ uncorrupted symbols, the probability of decoding failure is at most δ. This implies that $|\mu - \delta| = n\gamma$, and since Π is semantically secure, the advantage of $\mathcal{A}$ in winning the game is at most negligible.

Note that the above assumes that the randomness used to encode m is true randomness. The actual definition of Encode actually uses the pseudorandom generator G. By definition of a PRG, for any PPT algorithm Alg running in time t_G,

$$|P[x \leftarrow U_\lambda; \mathsf{Alg}(1^\lambda, x) = 1] - P[s \leftarrow U_\lambda; x \leftarrow G(1^\lambda, s); \mathsf{Alg}(1^\lambda, x) = 1]| = \gamma'$$

where U_λ is the uniform distribution over strings of length λ and γ' is negligible. Thus, by replacing true randomness in Encode with the output of G, the advantage of $\mathcal{A}$ increases by at most γ'. Hence, the advantage of $\mathcal{A}$ in causing a decoding failure is $\gamma n + \gamma'$, which is negligible.

Finally, if we allow $\mathcal{A}$ to corrupt symbols instead of just erasing, then the result still holds. That is, corrupting gives $\mathcal{A}$ no additional advantage. To see this, note that $\mathcal{A}$ must output at least $(1+\varepsilon)k$

uncorrupted symbols for a decoding failure to count as a win in ChannelExp. If any corrupted symbols are accepted by Decode (i.e., a MAC is successfully forged), then their acceptance *cannot* increase the probability of decoding failure. Rather, forgeries will *decrease* the probability of decoding failure by giving the LT-decoder more code symbols to use in decoding and thereby increase the coverage of message symbols by code symbols. This directly undermines the effort to cause a decoding failure. Hence, it is to $\mathcal{A}$'s advantage to only erase symbols. □

The following theorem summarizes the security of our scheme. The result follows directly from the two lemmas above.

Theorem 4.1. *Let $\mathcal{LTS}$ be a $(k, \delta, \mathcal{D}, \varepsilon)$-*Falcon *code utilizing an existentially-unforgeable MAC scheme M, a semantically secure cipher scheme Π, and a secure PRG G. Then, for all sufficiently large k, for all PPT $\mathcal{A}$, and for all feasible p, where $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$, we have that $\mathsf{Adv}_{A,\mathcal{LTS}}(\pi, p)$ is negligible in λ.*

Proof. Suppose not, then there exists a PPT adversary $\mathcal{A}$ such that, $\mathsf{Adv}_{\mathcal{LTS},\mathcal{A}}(1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$ is non-negligible in λ and k, call this advantage γ. $\mathcal{A}$ can win the game ChannelExp by causing a decoding failure (Decode outputs $\perp$) or a decoding error (Decode outputs the incorrect message). Since these are mutually exclusive events, we can consider them separately. By Lemma 4.1 and Lemma 4.2 we know that each event happens with negligible advantage. Hence γ is negligible. □

Note that the above theorem and lemmas apply to *non-systematic* Falcon codes. With a systematic code, $\mathcal{A}$ knows that the first k code symbols are equal to the input symbols. $\mathcal{A}$ can then make corruptions that are decidedly *non-random* against these symbols. That is, $\mathcal{A}$ has some a priori knowledge of the underlying encoding graph, and can adjust its strategy accordingly. Though we believe our construction is secure in the systematic case, we leave it to future work to prove this.

For our scalable FalconS codes (both with an explicit permutation and without), the security of the scheme follows from the results of Lipton's work on scrambled codes in [98]. In particular, the random permutation of the symbols ensures that the erasures and errors are uniformly distributed among the blocks and within each block as well, which is the definition of a random channel. Note that in this model, the adversary $\mathcal{R}_\gamma$ erases up to a γ-fraction of symbols (or a γ-fraction of batches) rather than erasing with probability γ. Recall that each block has $N = \frac{1}{1-(1+\tau)\gamma}(1+\varepsilon)k$ symbols with a corruption-tolerance parameter of τ.

Theorem 4.2. *Let $\mathcal{LTS}_{bN}$ be a $(k, \delta, \mathcal{D}, \varepsilon, \gamma, \tau, b)$-*FalconS *code that uses an existentially-unforgeable MAC scheme M, a semantically secure symmetric cipher Π, a secure PRG G to generate the permutation, divides the input into b blocks, and generates N symbols per block. Then, for all sufficiently large k and for all PPT $\mathcal{A}$ that corrupt up to a γ-fraction of symbols, letting $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$, we have $\mathsf{Adv}_{\mathcal{LTS}_{bN},\mathcal{A}}(\pi, \gamma)$ is negligible in λ.*

Proof. As shown in [98], the random permutation applied to the code symbols combined with perfectly secure encryption reduces $\mathcal{A}$ to a random channel. The use of G to produce the pseudorandomness to generate the permutation gives $\mathcal{A}$ at most a negligible advantage. Similarly, the semantic security of Π hides all but a negligible amount of information in the code symbols and gives $\mathcal{A}$ at most a negligible advantage over a random erasure channel. Finally, the existential unforgeability of M ensures that $\mathcal{A}$ can produce a decoding error with at most negligible probability. □

Finally, for our randomized Falcon codes, we utilize the main Falcon code as a subroutine and then use a strong PRG to select the block to produce the next symbol. The security of this scheme reduces to that of the PRG and the main Falcon code to get the following theorem. This result, however, does not follow directly from Lipton's work as above, since the symbols are output *in order from each block*. Rather, the PRG ensures that, when receiving symbols, with high-probability, after receiving $O(b[\log b + (m-1)\log\log b])$ uncorrupted symbols we can decode successfully. Using the main Falcon code ensures that the adversarial corruptions by $\mathcal{A}$ using a priori knowledge of the scheme—$\mathcal{A}$ knows that the first symbols in the stream are the first symbols encoded by the individual blocks—does not give $\mathcal{A}$ a significant advantage.

Theorem 4.3. *Let $\mathcal{LTS}_R$ be a $(k, \delta, \mathcal{D}, \varepsilon, b)$-*FalconR *code that uses an existentially-unforgeable MAC scheme M, a secure PRG G, a semantically-secure symmetric cipher Π, a secure $(k, \delta, \mathcal{D}, \varepsilon)$-*Falcon *code $\mathcal{LTS}$, and divides the input into b blocks. Then, for all sufficiently large k and for all PPT admissible $\mathcal{A}$, and for all feasible p, where $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon)$, we have that $\mathsf{Adv}_{\mathcal{LTS},\mathcal{A}}(\pi, p)$ is negligible in λ.*

Proof. First, we assume that any PPT $\mathcal{A}$ cannot distinguish between MACs and ciphertexts produced by different keys. That is, any PPT $\mathcal{A}$ cannot tell that $\mathsf{Mac}(k_1, m_1)$ and $\mathsf{Mac}(k_2, m_2)$ where produced using different keys, similarly for ciphertexts. While this is a limitation on the MACs and ciphers used, any symmetric cipher cipher or MAC used in practice will satisfy this requirement.

Suppose we have a PPT adversary $\mathcal{A}$ who can win $\mathsf{ChannelExp}_{\mathcal{LTS}_R,\mathcal{A}}$ with non-negligible advantage. Then we will construct $\mathcal{A}'$ who will break $\mathcal{LTS}$ with non-negligible advantage. First, assume that G produces perfect randomness. Construct $\mathcal{A}'$ as follows. $\mathcal{A}'$ generates keys k_{enc} and k_{mac} and runs $\mathcal{A}$ as a subroutine with parameters $\pi = (1^\lambda, k, \delta, \mathcal{D}, \varepsilon, b)$. $\mathcal{A}'$ then simulates the $\mathsf{FalconR}$ encoder by instantiating b separate instances of $\mathcal{LTS}$ and encoding each query of $\mathcal{A}$ appropriately.

Eventually, $\mathcal{A}$ outputs its challenge message m. $\mathcal{A}'$ selects one of the blocks at random (call it B) and then queries it to the Falcon oracle $\mathcal{O}_B$ (note, $\mathcal{A}'$ skips the query phase entirely). $\mathcal{A}'$ then initializes the encoders for the other blocks as before. $\mathcal{A}'$ then responds to $\mathcal{A}$'s symbol requests by either replying with a symbol from one of its encoders or, if the symbol is in B, sends it to $\mathcal{O}_B$. Eventually $\mathcal{A}$ outputs the code symbols $(\sigma_1, \ldots, \sigma_n)$ for the challenge message. $\mathcal{A}'$ selects the symbols that are for B, call these $(\sigma'_1, \ldots, \sigma'_{n'})$ where $n' \leq n$, and outputs only those as its corruption of B. Note that, with high-probability, $n' \geq (1+\varepsilon)k$, call this probability p. If $n' < (1+\varepsilon)k$, then $\mathcal{A}'$ fails.

Note that if $\mathcal{A}$ succeeds, then at least one of the blocks in the decoded message will either fail to decode or have a decoding error. If $\mathcal{A}'$ guessed B correctly, then $\mathcal{A}'$ succeeds as well. Hence, if $\mathcal{A}$ has advantage ϵ in succeeding, then $\mathcal{A}'$ succeeds with advantage at least $p\epsilon/b$ (and at most ϵ). Since we assume Falcon is secure, ϵ is negligible and so is the advantage of $\mathcal{A}'$. If we change G to use pseudorandomness instead, then the advantages of $\mathcal{A}$ and $\mathcal{A}'$ increase only by a negligible amount. □

4.7 Experiments

In this section we detail several experiments performed that demonstrate the practicality of our constructions. The experiments were run on two machines: one with "abundant" resources and the other with more limited CPU power and RAM. We use these two machines to measure the raw speed and efficiency of our schemes without constraints and to show that our scalable schemes do achieve better performance, especially when RAM becomes scarce.

The majority of the experiments were performed on the powerful machine with two 2.6GHz AMD Opteron 6282SE with 16 cores each and 64GB RAM running 64-bit Debian Linux with the 3.2.0 kernel and gcc version 4.7.2. The "resource-constrained" benchmarks were run on a 2.6GHz, Intel Core i5 processor with 4GB of RAM running 64-bit Arch Linux with the 3.14.19 kernel and gcc version 4.9.1. All implementations are a mix of C and C++ and are single threaded unless otherwise specified—the parallel implementation (detailed below) utilized pthreads—and were compiled with

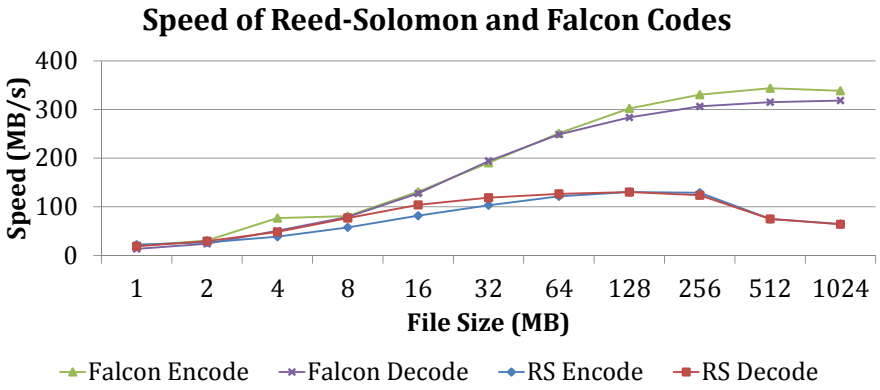


Figure 4.5: The encoding throughput of Falcon compared to RS codes.

the '-O2' optimization flag. All schemes use the Salsa20 stream cipher (see [15]) as the secure PRG, SFMT as the fast/insecure PRG (see [141]), and PBKDF2 for key derivation [74]. The Jerasure v2.0 library [71] was used for the Reed-Solomon erasure code.

We use authenticated encryption for both confidentiality and authentication: specifically, we use AES in Galois/Counter Mode (GCM) provided in the OpenSSL library (version 1.0.1e and 1.0.1i on the Opteron and Core i5 machines, respectively). An alternative implementation would be to use AES in counter mode and pair it with a fast MAC, such as VMAC [93]. In some rough tests on the Core i5, we found the AES-VMAC combination to be the fastest for symbols more than 2KB in size and AES-GCM to be faster for smaller symbols.[35] The largest symbol size we consider is 4KB, with almost all tests run on symbols 1KB or smaller. Thus, for simplicity, we use AES-GCM in all tests. We did not use batching and, hence, encrypted and MACed each symbol individually.

The algorithms we benchmark are Falcon Raptor codes: we combine our authenticated LT code with a precoding step (where an erasure code is applied to the input data) to give us a secure Raptor code. Raptor codes are among the most efficient erasure-correcting codes available and we show below that our authenticated raptor codes themselves achieve high efficiency. Our implementations of both Falcon codes and Raptor codes are based on the `libwireless` code written by Jonathan Perry [126]. We used an LDPC-Triangle code as the precoding step using the implementation from [155]. Unless noted otherwise, all encoding and decoding was performed with 1KB symbols and adding 25% redundancy and the numbers given are an average of 10 trials.

[35] For example, on a 16KB input, AES-GCM achieved 2.4 cycles-per-byte (cpb) while AES-VMAC achieved 1.7cpb; for a 256 byte input, AES-GCM was 7.6cpb and AES-VMAC was 14.0. All rates include key setup.

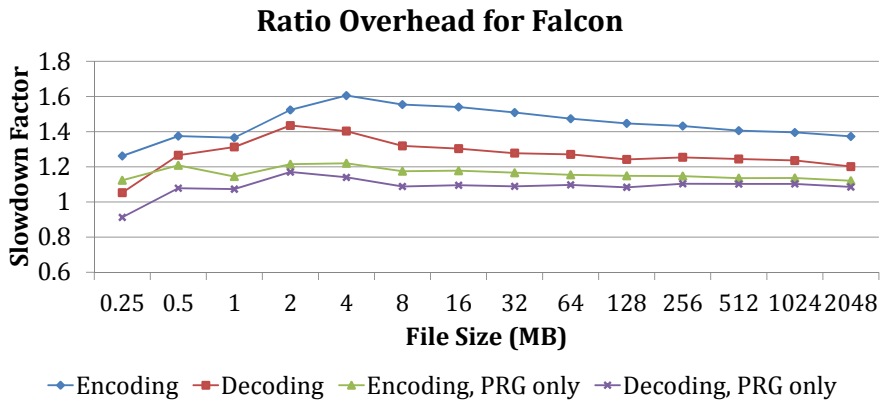


Figure 4.6: Slowdown of Falcon versus an insecure Raptor code (no crypto).

Main Scheme Falcon. First, we compare our Falcon scheme to that of Reed-Solomon codes (RS codes), which is the de facto standard for encoding a file to withstand adversarial corruption. In Figure 4.5, we see the encoding and decoding speeds for Falcon versus an RS code on various file sizes when run on the Core i5. For Falcon, the number of symbols was held constant at ≈ 10000 for all files with symbol sizes ranging from 16 bytes to 128KB. The RS encoder utilized a systematic erasure encoding with $k = 204$ and $n = 255$ (for 20% redundancy) over $GF(2^8)$ utilizing striping.[36] This allows us to quickly detect any tampering with the symbols and discard any that are corrupted and correct up to $n - k$ errors, the theoretical maximum.[37] As is clear from the graph, our scheme can achieve high throughput, reaching over 300MB/s for both encoding and decoding, and is several times faster than the RS encoder. Note that for inputs larger than 16MB, the throughput of Falcon can saturate a 1Gb network link.[38]

In Figure 4.6, we see a comparison of Falcon against an insecure Raptor code where no cryptography was used (i.e., no encryption, MAC, or secure PRG). The numbers shown are the average of 50 trials. The simple scheme is generally between a factor of 1.25 and a factor of 1.6 slower than the (completely) insecure scheme, and is usually less than 1.5 times slower. Note that the overhead from the use of the secure PRG results in a slowdown of approximately 10-15% (shown in the lower two lines). For larger files, the percent overhead from the cryptography declines as the LT and precode

[36] Striping is a technique where, instead of performing field operations over large symbols, the file is divided into symbols over a much smaller field (while using the same k and n) and encoded in small "batches" or "stripes." The small symbols are then grouped together to produce the large, output symbols. This can increase both encoding and decoding speeds.

[37] It is possible to use list-decoding for RS codes instead of MACs to correct up to $n - k$ errors, but list decoding algorithms, even the best ones, are many times slower than simply computing a MAC.

[38] The slowdown of the RS code for large files is from an increased miss-rate in the L3 cache: going from $\approx 5.8\%$ to $\approx 11.8\%$ when moving from 256MB input to 512MB input, as measured by the `cachegrind` tool of `valgrind` [115].

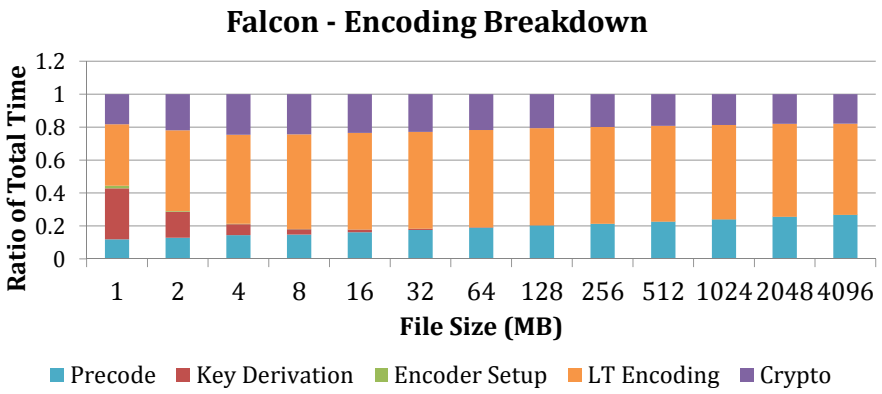


Figure 4.7: Ratio of the total time spent in each part of encoding process for Falcon.

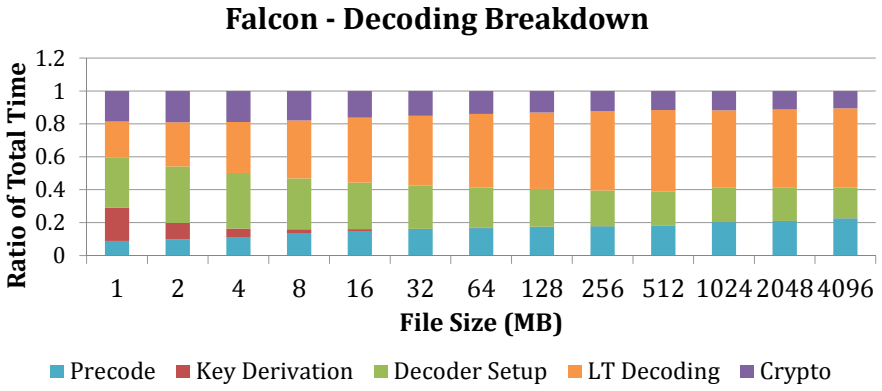


Figure 4.8: Ratio of the total time spent in each part of decoding process for Falcon.

encoding takes a larger percentage of the total encoding time. (This is also seen in Figures 4.7 and 4.8.) The "increase" in speed for decoding a 256KB file when using the secure PRG instead of an insecure one is due to environmental noise and the very short encoding times. The difference in the encoding times for the secure and insecure PRG is around 20 to 30 microseconds.

Figures 4.7 and 4.8 show a breakdown of the time to encode and decode various file sizes by the Falcon scheme. For encoding, the cost to create the encoder is negligible compared to the remainder of the encoding while for decoding creating the decoder can take a significant amount of time. This is due to the decoder pre-computing much of the LT-decoding information (i.e., node degrees and neighbor sets) to achieve faster overall decoding. Note that in both cases, as files become larger, the overhead from the cryptographic "post-processing" (i.e., encryption and MAC computation) decreases as a percentage of the total time.

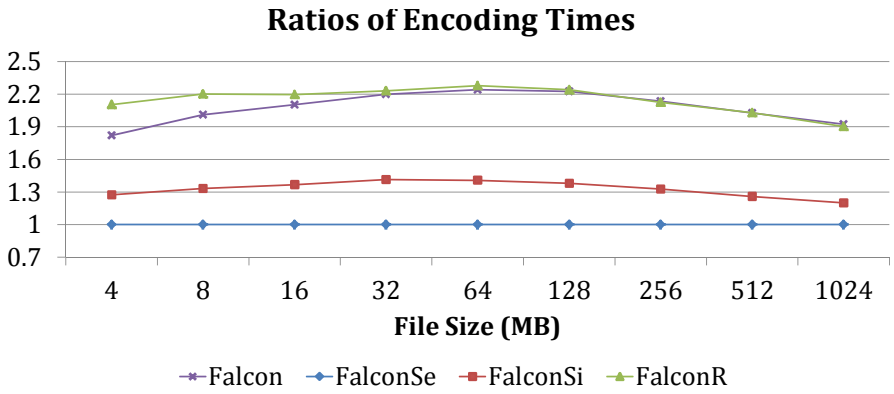


Figure 4.9: Ratios of the encoding times of various file sizes by all schemes. Files were encoded with 64 blocks and 128 byte symbols.

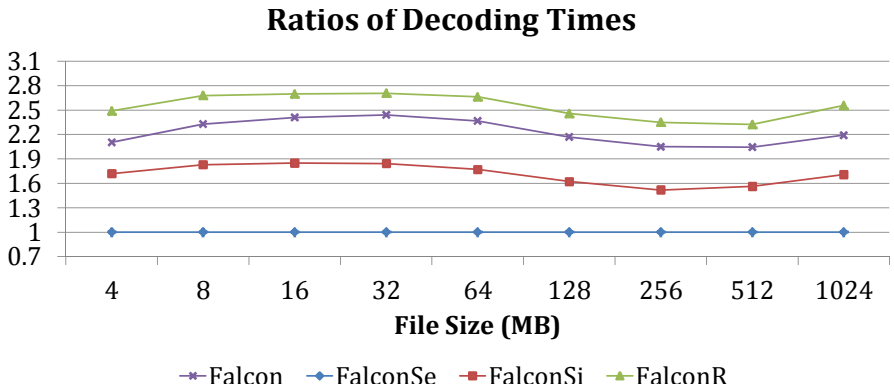


Figure 4.10: Ratios of the decoding times of various file sizes by all schemes. Files were encoded with 64 blocks and 128 byte symbols.

Scalability: Abundant Resources. In Figure 4.9, we can see that the FalconSe and FalconSi and schemes are, indeed, more scalable than Falcon. Note that FalconR performs worse than Falcon on smaller inputs and almost identically on larger inputs. (The difference in performance of these two is evident in Figure 4.13.) Falcon is slower than FalconSe and FalconSi primarily because it utilizes a secure PRG in the LT-coding and it has worse locality of reference: when performing the LT-coding, it combines symbols from across the entire file rather than just a segment of it. FalconSe is more efficient than FalconSi due to better locality of reference both for code and data since FalconSi continually switches between the encoders for each block. FalconSe encodes one block at a time and thus keeps less data resident at any given time, better utilizing caches. FalconR is the least efficient for three reasons: (1) it undermines locality by switching between encoders, (2) it generates more symbols than all other schemes (cf. Figure 4.11), and (3) it uses a secure PRG in the LT-coding, in contrast

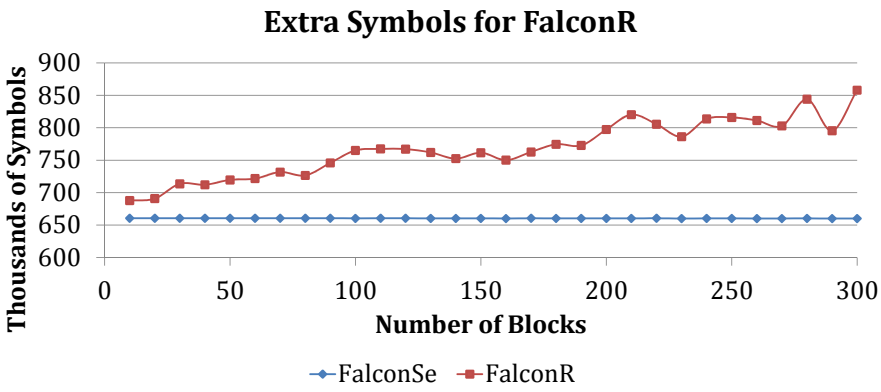


Figure 4.11: Number of symbols generated by FalconSe and FalconR versus the number of blocks.

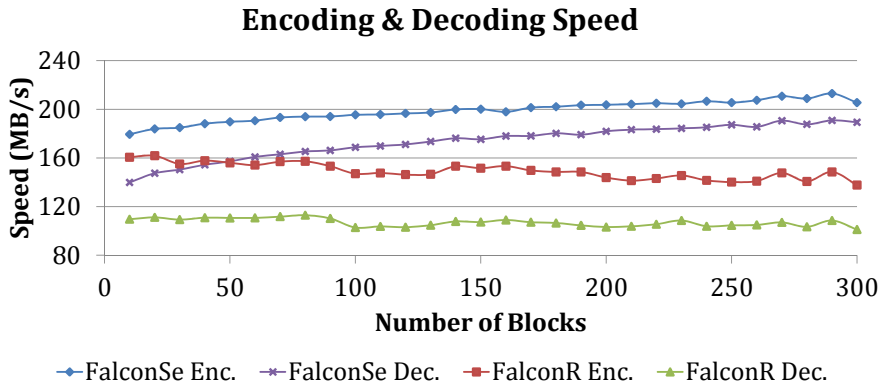


Figure 4.12: Encoding/decoding speeds of FalconSe and FalconR.

to the FalconS schemes. The benefit of using FalconR is shown in Figure 4.13 where it performs well when RAM is limited.

FalconR requires that we generate more symbols than FalconSe to ensure that we have enough to properly decode each block (see Section 4.5.3). FalconSe, in contrast, generates exactly the number of symbols required. In Figure 4.11 we can see the growth in the number of symbols needed by FalconR as compared to FalconSe. The input was a 2GB file and both schemes used 4KB symbols. The FalconR encoder works by having each block require a minimum number of symbols m to be produced and then generating symbols until each block has at least m symbols. When adding 25% redundancy, the encoder simply requires $(1.25)m$ symbols for each block. As mentioned in Section 4.5.3, the paper [117] proves that for a small number of blocks, the expected number of symbols output is $O(bm)$. This is evident in in the figure since the number of symbols produced is close to optimal for $b < 40$. However, for larger b, the expected number produced is $O(b[\log b + (m-1)\log\log b])$, visible in the growing number of symbols generated as b increases.

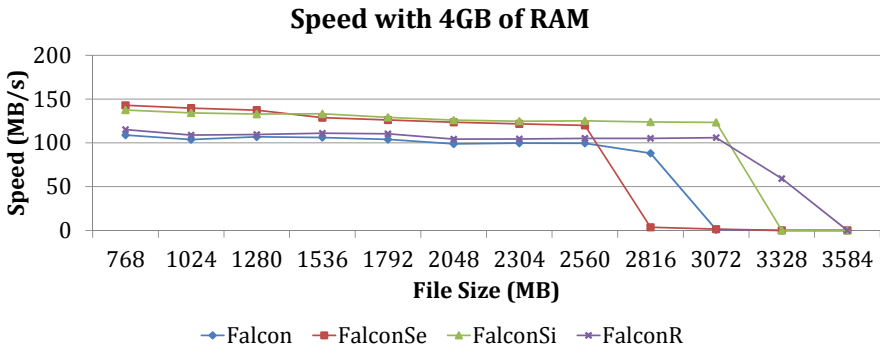


Figure 4.13: Encoding throughput of the different schemes in a more limited environment. Scalable schemes used 64 blocks.

Figure 4.12 compares the speed of FalconR to FalconSe on the same input as Figure 4.11. As the number of blocks increases, the size of each block decreases, allowing each individual block to be encoded faster. This is evident in the increasing speed of encoding and decoding for FalconSe. FalconR does not display this behavior: rather, its speed decreases since it must generate more symbols as the number of blocks increases as seen in Figure 4.11.

Scalability: Strained Resources. The benchmarks with strained resources were run on a 2.6GHz, Intel Core i5 processor with 4GB of RAM running Arch Linux. The tests compare the throughput of the encoder for each schemes given in Section 4.5, on files ranging from 768MB to 3.5GB in increments of 256MB. In each test, we read in the entire file prior to encoding, but did not include this time in the measurements.

In Figure 4.13, Falcon does reasonably well compared to the other schemes until the data and the overhead from the encoder take up too much RAM. As soon as Falcon needs to use external memory the throughput plummets and declines to almost zero. Comparing to Falcon, we can see that FalconSe fairs poor for files above 2.5GB since it must buffer all symbols before emitting them as it must have all symbols available before permuting them (while the other schemes can output symbols immediately) and its throughput goes to essentially zero. FalconSi and FalconR, on the other hand, fare better and can encode files larger than Falcon and provide good throughput until just after 3GB when they start heavily swapping.

Parallelism. Each of our schemes allows for some level of parallelism and can take advantage of multiple processing cores on a CPU. Falcon allows for parallelism at the symbol level: once the degree

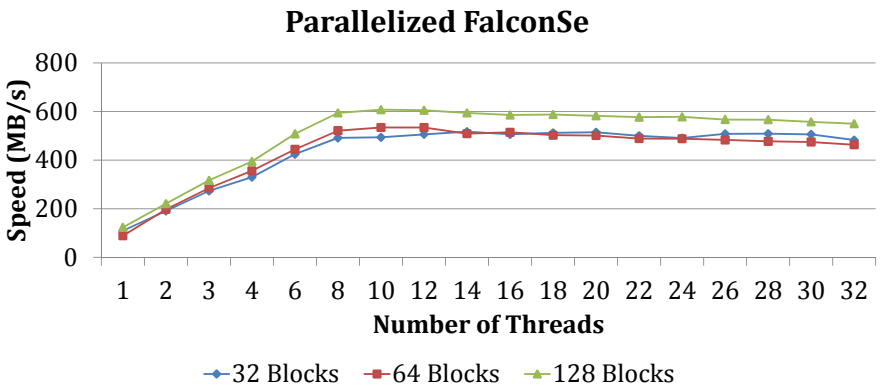


Figure 4.14: Throughput of parallelized FalconSe.

and neighbor set of a given symbol has been computed, it can be encoded independently of the other symbols. Each of the scalable schemes are trivially parallelizable at the block level, which we implemented and measured.

Figure 4.14 shows the results of running a multi-threaded version of FalconSe on the abundant-resource machine, using a 2GB file as input with various numbers of blocks and threads. In each case, the performance increases until there are 8 threads, and then it levels off.[39] This plateau is due to the AES-GCM encryption in the OpenSSL library reaching its peak throughput for the machine. In particular, we measured the performance of AES-GCM on the Opteron 6282SE for 1024 byte inputs and a peak throughput of approximately 620MB/s including key setup.[40] The cost of the remainder of the encoding process (e.g., XORing symbols, allocating and deallocating memory, etc.) accounts for the difference between the measured throughput of 500-600MB/s and the theoretical maximum of 620MB/s. The slow decline in performance as the number of threads increases is (at least partly) due to increased contention for the data caches. In particular, the fraction of data references that miss the last level cache increase from 19% to 23.8% when increasing from 8 to 32 threads, as measured by Linux's `perf` utility, averaging over 20 runs with a standard deviation between 0.1% and 0.7% depending on the number of threads—though, usually, it was around 0.3%. Thus we can see that the scalable Falcon encoders can, indeed, achieve very high performance using multiple threads and are limited primarily by the efficiency of the cryptography used.

[39]The version of `libc` on the machine is `eglibc` version 2.13 from 2011. The implementation of `malloc` and `free` in this version do not scale to multiple threads and, indeed, multiple threads can easily cause each other to block. To avoid this we used the `lockless` memory allocator library [68].

[40]The test simply set up a fixed key and encrypted the same 1024 byte 'buffer 100000 times and measured the total elapsed time.

4.8 Previous Work

Computationally-bounded Channels. In this work, we prove our constructions secure against a computationally-bounded adversary. This is more limited than the general, information-theoretic approach generally taken in information theory. But, since attacks in the real world are restricted to feasible computations, this limitation gives a more accurate model of the world. This approach was first proposed in [98], in which Lipton introduces and formalizes the concept of a computationally-bounded adversarial channel (this was further developed in the unpublished manuscript [55]). In this, he shows via a simple construction (called "code scrambling"), that any code that has success probability q over the binary symmetric channel with error probability p, has success probably q over *any* (computationally-bounded channel) with error probability p. The construction works by simply generating and applying a random permutation of n elements to the code word than then adding in a random pad. This requires $\Theta(n \log n)$ random bits, but by assuming the existence of a PRG with security parameter λ, then the construction only needs $O(\lambda)$ truly random bits (where $\lambda \ll n$). He also presents a construction that divides the input into blocks of size $a \log n$ (for a suitably chosen a), independently encodes them, and then scrambles them all together. Before the scrambling, the construction could handle $O(\log n)$ errors in the worst case, but after it can handle $O(n)$ errors.

In [95], Langberg defines the notion of a private code: i.e., a code that takes a secret key as a parameter. In this work, he proves that any code over the binary symmetric channel (with error probability p) can be turned into a private code over an adversarial channel that corrupts at most pn symbols. This is done, as in [98], via random permutation over the code symbols. The difference between [95] and [98], is that the former does not make any cryptographic assumptions (so the key must be $\Theta(n \log n)$ bits). In addition to this simple construction, Langberg proves a lower bound of $\Omega(\log n)$ on the number of random bits needed to be secure against an adversarial channel. Additionally, he provides a construction that meets this lower-bound (i.e., requires only $O(\log n)$ random bits). This construction works via a careful partitioning the space of code words for a list-decodable code. Decoding is performed with a maximum-likelihood decoder.

In [111], Micali et al. describe a game for a stateful, computationally-bounded adversarial channel and provide a construction that is secure against that channel. They describe a game—modeled after cryptographic games—involving multiple rounds where the channel chooses a message to be encoded, corrupts the encoding, and gives the result to the receiver. The adversary wins if at any point the

decoder decodes to an incorrect message. The construction they provide is essentially a combination of digital signatures and list-decoding. Specifically, they sign a message before encoding it with an efficiently list-decodable code. When decoding, the received codeword is list decoded and the signature is used to disambiguate the list and achieve unique decoding. This is essentially the same construction found in the earlier paper [106] by Lysyanskaya et al. that applied the same technique to authenticate a group of packets. In both works, adversarial success reduces to forging a signature.

In work combining cryptography and error correcting codes, Luby and Mitzenmacher in [104] describe a "verification decoding" algorithm based on belief propagation for LDPC codes. They assume that a corrupted packet takes on a uniformly random value. The decoder performs two steps repeatedly: (1) if the sum of all neighbors of a parity node is 0, then mark all nodes as verified, (2) if a parity node has a single unverified neighbor, then it is marked as verified and its value is set to the sum of all other neighbors. With this decoder, they are able to correct errors over the binary symmetric channel. Building on this, they independently rediscover the result of Lipton [98] and reduce an adversarial channel to the binary symmetric channel. They accomplish this by applying a random linear transformation to all code symbols and then permuting them. (The same authors also use this technique in [114] to reduce adversarial corruptions to random ones for a code based on invertible Bloom lookup tables; see [114] for details.)

Smith has also worked to protect against adversarial errors in [150], where he sought to use only a few random bits and avoid any unproven computational assumptions (e.g., the existence of a cryptographically strong PRG). In particular, he draws a random permutation from a t-wise independent family, where $t = o(n)$.[41] Moreover, this is accomplished using $n + o(n)$ random bits instead of the previous best of $\Theta(n \log n)$. The basic construction is the same as that of Lipton in [98], where a plain ECC is applied to a message, and then random permutation is applied to the code word, scrambling it. The resulting codes are capacity approaching and correct pn errors with probability exponentially close to 1. Our work makes additional assumptions (i.e., the existence of secure MACs, encryption, and PRGs) and allows us to use far less randomness, i.e., we need $\lambda \ll n$ bits.

Guruswami and Smith in [59] present several results for codes against computationally-bounded adversaries that corrupt up to a p fraction of the code symbols. First, they provide constructions for codes that approach channel capacity for additive (i.e., oblivious) channels. Second, they give polynomial time list-decodable codes with optimal rate for log-space adversaries. Their construction

[41] Such a permutation can be generated using $O(t \log n)$ bits using a specific (non-cryptographic) PRG.

for list-decodable codes can be extended to handle an adversary with n^c space for any fixed $c > 1$ (but c must be fixed beforehand). For the first two constructions, they only assume the existence of one-way functions, and for the latter they assume the existence of PRGs that can fool circuits of size n^c. In their constructions, the sender and receiver do *not* have pre-shared secret unknown to the adversary. They hide the secret randomness in the message itself such that the adversary (with only log-space) cannot locate it, but the receiver (who has super-logarithmic space) can.

Secure LT Codes. The work in [91] by Krohn et al. provides an efficient construction for online verification of erasure encoded symbols produces by a rateless code, such as Raptor or Online codes, for content distribution in peer-to-peer networks. Their adversary is one that sends spurious blocks to clients requesting a given file. The authors use homomorphic hashing—via exponentiation in a group of prime order—on the message symbols to produce a succinct digest of the file for verification. Hashes for code symbols can be calculated by multiplying together the hashes for the corresponding message symbols. Since the hashes for a file can be quite large—for example, an 8GB file would have 64MB of hashes—they also give a recursive hashing scheme that can greatly reduce the number of hashes sent over the wire. The security of the scheme reduces to the difficulty of computing discrete logarithms in the group. Their scheme has several "knobs" that allow adjusting the trade-offs between verification speed, amount of authentication information sent, and how quickly a malicious server can be identified (i.e., how many blocks must be received before maliciousness is detected). In comparing with our scheme, we note that we allow the adversary to have access to entire file and all of authentication tags while they assume reliable delivery of, at least, the root hash computed over an encoded file. The authors observe the possibility of targeted-erasure attacks (which they call "distribution attacks"), but they leave resilience to such attacks to future work.

In [158], Tartary and Wang build on the work of Lysyanskaya et at. in [106], and use LT codes as part of a multicast authentication scheme over an (α, β)-network.[42] Specifically, they apply an LT code to a set of n packets and generate N code symbols. Each symbol is then hashed along with the indices of its neighbors. The hashes are signed and the signature is appended to their concatenation. The N hashes and signature are then encoded with an $[N, \alpha N]$ Reed-Solomon code. Each code symbol c_i from the LT code has the indices of its neighbors appended as well as the i-th code symbol of the encoded hashes and signature. To decode, they apply list decoding to encoded hashes and use

[42] Recall that an (α, β)-network is a network work, when sending n packets at least an αn packets will arrive unscathed, and at most βn will arrive total.

the signature to achieve unique decoding (as in [106] and [111]) and recover the hashes which are used to detect corruption of the LT code symbols. The authors only consider attacks that corrupt the data and do not consider the targeted-erasure attack.

Codes & Cryptography. There are several examples of schemes that combine cryptography and ECCs together. An early such example is by Krawczyk in [88] where he computes the hashes of erasure encoded symbols and then uses an error-correcting code to encode each hash. The shares of each encoded hash are then distributed with the symbols; this technique is an application of distributed fingerprints [87], also by Krawczyk, and is used to detect corruption of code symbols. An example where cryptography is used in the heart of an ECC is by Perry et al. in [127] (further developed in [128]) where they present a new rateless error correcting code called a *spinal code*. Spinal codes work by breaking the input message of n bits into blocks of k bits and then applying a random hash function to each block to produce a real number in the interval $[0, 1)$ (called a *spine*). The encoder then makes passes over the spines and applies a deterministic function to map a portion of each spine to an output bit.

Computational Locally Decodable Codes. Chandran et al. in [26] define locally-updatable, locally-decodable codes (LULDCs) and present constructions that are constant rate while allowing for local updates to an encoded message and providing local decodability. Update operations have a locality of $O(\lambda \log k)$ and a read locality of $O(\lambda \log^2 k)$. The constructions are secure against computationally-unbounded adversaries, but the adversaries are limited such that they do not corrupt "too many" of the recently updated bits: that is, only a few of the new bits can be corrupted, but many of old bits can be. This is accomplished by having a logarithmic number of buffers arranged in a hierarchy and exponentially growing in size. Updates start in the lowest (and smallest) levels and percolate up to the higher levels as more updates are made. Each level is encoded with a locally-decodable code. Against computationally-bounded adversaries, they can detected arbitrary errors (via MACs) and can correct a limited class of corruptions.

In [119], Ostrovsky et al. present several constructions for private locally decodable codes against computationally-bounded adversaries. By assuming the existence of one-way functions, they construct asymptotically good locally decodable codes over a binary alphabet. They can correctly decode any bit after querying $\omega(\log^2 \lambda)$ bits in a codeword, with probability greater than $1 - \lambda^{-\omega(1)}$. If

the sender and receiver have a shared state (e.g., a public counter), then their query complexity is $\omega(\log \lambda)$. Furthermore, they show that $\omega(\log \lambda)$ is necessary to achieve negligibly small probability of incorrectly decoding.

Proofs-of-Retrievability. A proof-of-retrievability (PoR), first defined and explored by Juels and Kaliski in [72], is scheme that makes heavy use of cryptography and error correcting codes to secure remote storage. PoRs consist of encoding, decoding, and audit protocols, the last of which is used to detect data corruption. Intuitively, a PoR guarantees that if the adversary does not corrupt too much, then the damage can be repaired; conversely, if the damage cannot be repaired, then it will be detected with high probability.

There have been several follow-up works to [72]. Shacham and Waters in [145] present two PoRs: the first is secure in the random oracle model and has short audit query and responses, while the second is secure in the standard model and also has short audit responses but longer queries. The work of [20] by Bowers et al. combines Reed-Solomon codes and universal hashing to produce secure, homomorphic MACs over the data that gives an efficient audit protocol. In addition, the adversary is a *mobile adversary* that may (over time) corrupt all servers but at any given time it only has up to a constant fraction compromised. PoRs are further explored in [21] where Bowers et al. provide rigorous theoretical framework for designing PoRs and provide improved variants of the constructions in [72] and [145]. They also define an *adversarial error correcting code*, which is a code that can tolerate some amount of adversarial corruption, but their definition is limited to block codes and cannot be generalized to rateless codes. Moreover, their security game for the codes explicitly forbids the adversary from causing a decoding failure which we allow here. In [142], Sarkar and Safavi-Naini present a PoR that utilizes raptor codes: specifically, they apply an erasure code to the data and apply a homomorphic authenticator. The audit protocol then performs (standard) LT encoding allowing both an unbounded number of challenges to be created and efficient encoding and decoding.

Much early work for PoRs assumed that the encoded data was static, and the resulting schemes could not be readily extended to the dynamic case. Shi et al. provide an efficient PoR for dynamic data in [147] where they use a hierarchical log structure, where each level is erasure encoded, combined with Merkle trees and MACs. They achieve efficiency that is logarithmic in the number of blocks (multiplied by the security parameter) for both bandwidth and server computation with audit costs also logarithmic in the number of blocks and quadratic in the security parameter.

Network Coding. Ho et al. present in [65] an information-theoretically secure construction for random network coding where the adversary $\mathcal{A}$ is computationally-unbounded and controls a constant fraction of the network (but not all of it). Each packet is augmented with a polynomial hash of the packet, and then at each step in the network, all incoming packets are combined in a random linear combination. The decoder collects enough linearly independent packets to recover the input packets, checking the polynomial hashes against the decoded data and throwing out anything corrupted. Since $\mathcal{A}$ does not control the entire network, some of the packets are unknown to $\mathcal{A}$ and he is unable (except with small probability) to manipulate the packets to cause a decoding error. The probability of corruption detection is a tunable design parameter of the network.

Jaggi et al. in [70] provide information theoretically secure network codes that can tolerate byzantine nodes in the network. If the network has capacity c and the adversary can eavesdrop on all links and jam a z-fraction of the links, then their codes achieve a rate of $c - 2z$. If the adversary has limited knowledge, e.g., can only eavesdrop on a fraction of the links, then their codes have rate $c - z$. The also provide construction can also utilize a secret, noiseless side channel between the sender and receiver to facilitate decoding. In particular, the receiver uses list decoding to recover a list containing the correct decoding and then uses a hash of the original message (sent over the side channel) to disambiguate the list decoding.

Multicast Authentication. Lysyanskaya et al. couple signatures with list-decoding to achieve error correction beyond the unique decoding radius [106]. Specifically, they sign the input message and then apply an efficiently list-decodable code (e.g., a Reed-Solomon code). When decoding, the codeword is list-decoded and the signature on each entry in the list is checked and is used to select the correct message as the output. This technique of using signatures to achieve unique decoding from list-decoding over a computationally-bounded channel was independently discovered by Micali et al. in [111]. This latter work also developed a theoretical foundation for a computationally-bounded channel that attacks a block code.

Pannetrat and Molva combine error correcting codes with cryptographic hashes and signatures for efficient multicast authentication over adversarial erasure channels, where the channel can erase up to a constant fraction p of the packets [121]. Specifically, they break the packets up into blocks, hash the packets, sign the hashes, and then use a systematic erasure code to encode the hashes and signatures. The parity symbols generated from the encoding are then concatenated and divided

up, evenly, among the packets. This results in tens of bytes overhead per packet. Given $(1-p)n$ packets (where n is the number of packets per block), the hashes are computed for the packets and the parity symbols for the encoded hashes and signature are reconstituted. Since the erasure coding was systematic, the combination of $(1-p)n$ hashes with the reconstituted parity symbols is enough to recover all the hashes and signature. The signature is verified and the packets are all thrown out if the signature is invalid.

Karlof et al. define *distillation codes* in [76] for use in authenticated multicast. Essentially, distillation codes are an erasure code augmented with a one-way accumulator and a signature scheme. (For concreteness, they use a Merkle tree as the accumulator.) For the scheme, the input is "tagged" (e.g., signed or MACed) and then encoded. The encoded symbols are then hashed as the leaves in a Merkle tree. Each symbol is then sent with the neighbors on the path from the symbol to the root of the Merkle tree. When decoding, the received symbols are partitioned into groups based on the calculated root hash value. Each group is then erasure decoded, and for any successful decodings the tag is "validated" and all groups with an invalid tag are discarded. The decoder then randomly selects one of the remaining groups and outputs the decoding. This work is designed to allow for arbitrary errors in addition to "pollution" attacks, where the adversary injects extra (spurious) symbols into the stream. Their scheme is secure over a computationally-bounded channel, though this is not stated nor formalized.

In [158], Tartary and Wang build on the work of Lysyanskaya et al. [106] and use LT codes as part of a multicast authentication scheme. Specifically, they apply an LT code to a set of n packets and generate N code symbols. Each symbol is then hashed along with the indices of its neighbors. The hashes are signed and the signature is appended to them. These are then broken into k pieces and regarded as coefficients of a polynomial $p(x)$ in $\mathbb{F}_{2^r}[x]$. The polynomial $p(x)$ is then evaluated at the first N points of $\mathbb{F}_{2^r}$. Code symbol c_i then has the indices of its neighbors appended as well as the value of $p(i)$. In decoding, they apply Reed-Solomon list decoding to the polynomial values and use the signature to achieve unique decoding (as in [106] and [111]). Their security model only considers data corruption attacks does not take into account the targeted-erasure attack that we described earlier. This leads to a gap in their security proof since the probability of decoding failure δ for LT codes is computed based on the assumption of *random* erasures. That is, even if the adversary is not specifically trying to cause a decoding failure, because the corruptions are not necessarily random, we lose the guarantee that δ is an upper-bound on the probability of decoding failure.

Secure Storage. Cao et al. utilize LT codes, bilinear maps, and homomorphic MACs for a secure cloud storage scheme that allows for asymptotically efficient encoding and decoding, as well as repair of the data if any servers are lost using a designated repair server [25]. In this setting, they consider computationally-bounded adversaries, but only consider attacks against the data servers and not against the repair server. They take into consideration the targeted-erasure attack that we described earlier. Though, their solution was simply to check that all subsets of size k out of n code symbols can be successfully decoded and re-encoding the whole file if this was not the case. There is no analysis of the likelihood of this "decodability check" failing, nor any measurement of how long this check could take.

Non-malleable Codes. Looking at cryptography and codes, there are primitives known as *non-malleable codes*, first defined and constructed by Dziembowski et al. in [39]. The work was further developed by Faust et al. [40] and Cheraghchi and Guruswami [28]. Non-malleable codes seek to encode a message such that, if it is tampered with, it will decode to a message *unrelated* to the original message. They were designed to protect against malicious hardware tampering so that any adversarial manipulation results in a random internal state (rather than a maliciously chosen one).

Cheraghchi and Guruswami provide a construction for a non-malleable code that is superficially similar to our block, scalable construction [28]. Specifically, the message is encoded with a linear error correcting secret sharing scheme, then they divide a message up into blocks and apply a weak non-malleable code to each block. The blocks are then randomly permuted. In this way, if an adversary corrupts a small portion of the message, then the error-correction will fix the corruption. If, however, the adversary corrupts more than the ECC can handle, then the (weak) non-malleability of each block will cause the corruption to "blow up" and be detected. In contrast, we divide the input into blocks, apply a weak encoder to each block and then permute all the symbols. We parameterize the scheme so that we add enough redundancy to the block that the attacker can only "overwhelm" a block with negligible probability.

4.9 Conclusion

We introduced a new security model for analyzing fountain codes over computationally-bounded adversarial channels, and presented Falcon codes, a class of (block or rateless) authenticated ECCs that

are based on the widely used LT codes. Falcon codes are provably secure in our model while maintaining both practical and theoretical efficiency (including linear-time coding for Falcon Raptor codes). Their efficiency makes them a useful, general-purpose security tool for many practical applications such as secure and reliable data transmission over a noisy (and possibly malicious) channel and secure data storage.

CHAPTER **Five**

Conclusion

In this thesis, we detailed several novel tools that readily enhance the privacy, integrity, and fault-tolerance of cloud storage. We provided a simple, yet itself novel, architecture for increasing data integrity and fault-tolerance in cloud storage (without modifying the service provider), and we showed how our constructions can greatly improve the efficiency of even such a simple scheme.

The first tools we described were the Slow and Fast Squeeze ciphers, two new and provably secure ciphers that provide data compression in addition to privacy. These are the first provably secure schemes that combine together data compression with encryption. The algorithms themselves are derived from the well-known LZW compression algorithm [165] which allowed for efficient implementations of our schemes, demonstrated through a thorough experimental evaluation. We also provided a new definitional framework for proving the security of combined compression-encryption schemes and showed that it relates in a simple way to the standard definitions of security for ciphers. Moreover, we proved that our constructions achieve the strongest possible security in this model.

We then presented a new adversarial model for analyzing the error-correcting properties of certain codes when attacked by computationally-bounded adversaries. This model is more powerful than previously considered and is able to capture more real-world adversarial behaviors. We also provided two general constructions—which we call *authenticated error correcting codes*—that cryptographically enhance erasure codes to (provably) provide error correction when attacked by our powerful adversary, with only a small loss in code rate. The first construction combined list decoding with collision-resistant hash functions and digital signatures to correct errors more efficiently than previous list-decoding-based schemes (i.e., [106] and [111]). Our second construction utilized a non-malleable cipher, a

message authentication code, and a generic pseudorandom permutation to achieve the same (provably secure) error correction without using list decoding.

Finally, we introduced a novel family of error-correcting codes—called *Falcon codes*—that enhance LT codes (an efficient family of rateless erasure codes) by integrating a pseudorandom generator, semantically secure encryption, and a message authentication code into the LT code itself. These changes allow Falcon codes to withstand *adversarial* corruption of data and we proved—in our new adversarial model for rateless codes—that any computationally-bounded adversary can do no better (when attacking the code) than simply erasing symbols at random. Additionally, we provided two alternative *scalable* variants that can encode and decode large files more efficiently than the basic scheme. Falcon codes are a generic construction that can replace any LT code and immediately enhance the error tolerance and security of the scheme. For example, we used Falcon codes in the LT coding step of Raptor codes to produce an error correcting Raptor code and experimentally demonstrated its practicality (along with the scalable constructions).

APPENDIX **A**

Equivalence of Indistinguishability and Reseedable Indistinguishability

Here we prove the equivalence of normal indistinguishability for a pseudorandom generator and our notion of *reseedable-indistinguishability*. Intuitively, these notions seem equivalent since a distinguisher $\mathcal{D}$ could always generate random seeds for the PRG G and look at its output. Moreover, the distinguisher could also generate all the random bits it desires. So, reseeding the source of input bits should give only a negligible benefit in distinguishing. We prove this below.

Lemma (Indistinguishability Equivalience)**.** *Any (t, b, ε)-indistinguishable pseudorandom generator G is also a $(\varepsilon r^2, r+1, t-r)$-reseedably-indistinguishable pseudorandom generator, where a distinguisher may run in time t and request $r+1$ reseeding operations.*

Proof. The proof proceeds via a standard hybrid argument. Suppose we have a distinguisher $\mathcal{D}$ that makes $r+1$ re-seeding requests and can distinguish the output of G from a random oracle with probability at least δ. And suppose we have an input string s that may be the output of G under a random seed or a random string. $\mathcal{D}$ outputs 1 if it thinks its input was the output of G under random seeds and 0 otherwise.

Define H_i, where $1 \leq i < r+1$, be the distribution where the first i samples given to $\mathcal{D}$ are from G using a random seed for each sample, the $i+1$-th sample being the input string, and all remaining samples are random strings. Note that $\mathcal{D}$ can distinguish H_1 from H_{r+1} with probability at least δ. So, there exists some $1 \leq j < r+1$ where $\mathcal{D}$ can distinguish H_j and H_{j+1} with probability at least δ/r. We construct algorithm $\mathcal{A}$ to distinguish the output of G from random using $\mathcal{D}$ as a subroutine.

First, on input 1^λ and string s (where $s \leftarrow D_b$, with $b \xleftarrow{R} \{0,1\}$, $D_0 = U_n$ and $D_1 = \{G(s) | s \leftarrow U_\lambda\}$), choose a random index j between 1 and r (inclusive). Run $\mathcal{D}$ on input 1^λ. For the first j samples

given to $\mathcal{D}$, choose a random seed s_k and give $\mathcal{D}$ the output of $G(s_k)$. For the $j+1$-th query, give $\mathcal{D}$ the input string s, and for all remaining requests, give $\mathcal{D}$ a random string. $\mathcal{D}$ then outputs a bit b', and $\mathcal{A}$ outputs it as well.

Note that the input given to $\mathcal{D}$ is from either H_j or H_{j+1}. With probability at least $\frac{1}{r}$, the j selected is then one where $\mathcal{D}$ successfully distinguishes the distributions with probability at least δ/r. That is, $\mathcal{D}$ outputs 0 when its input is from H_j and 1 when its input is from H_{j+1} with probability at least δ/r (in each case). This means that $\mathcal{A}$ succeeds with probability at least δ/r^2. Since G is (ε, t)-indistinguishable, we have that it is also $(\varepsilon r^2, r+1, t-r)$-reseedably-indistinguishable.[1] □

[1] Note that we assume that generating a random string and generating $G(s)$ for a random seed both take constant time.

APPENDIX **B**

Application of Falcon Codes to Proofs-of-Retrievability

Bowers et al. [21] describe a framework for constructing proofs-of-retrievability (PoRs). Their framework includes two phases: (1) a challenge-response phase where the client sends challenges to a (possibly malicious) server to ensure (with high-probability) that at most an ϵ-fraction of the input has been corrupted; and (2) an extract phase that, given an adversary that corrupts an ϵ-fraction of the file, executes a series of challenges and responses that allow the client to extract the original file.

Their example construction utilizes two layers of error correcting codes: (1) an *outer code* applied to the original file, and (2) an *inner code* used in the challenge-response phase. In particular, the inner code ensures that the adversary corrupts at most an ϵ-fraction of the file (else, this excess corruption is detected), while the outer code ensures that the client can correct up to an ϵ-fraction of file corruption. The outer code must be what they term an *adversarial error correcting code* (AECC), which intuitively operates by turning a computationally-bounded adversary into a random one (as we also do). In [21], an $[n, k]$ ECC is defined to be a (β, δ)-bounded AECC if the probability that the adversary can produce two valid codewords that are at most βn apart is at most δ.

Falcon codes reduce adversarial corruptions to random erasures but do not precisely fit into the definition of an AECC [21]. Specifically, Falcon codes are rateless codes rather than block codes and so there is no fixed n. However, if we take the n-symbol block variant of Falcon codes (where exactly n symbols are output by the encoder), then we have that a $(k, \delta, \mathcal{D}, \varepsilon)$-Falcon code F is a $(1, n\varepsilon_{MAC})$-bounded AECC where ε_{MAC} is the probability of forging a MAC for the MAC scheme used in F (cf. Lemma 2 for Falcon). Thus, Falcon codes can be employed as an efficient outer code for a PoR. Moreover, Falcon does not require striping as in [21].

BIBLIOGRAPHY

[1] 3GPP. Multimedia Broadcast/Multimedia Service (MBMS); Protocols and codecs. Technical Report ETSI TS 26.346, 3rd Generation Partnership Project, September 2014. Version 12.3.0, Release 12.

[2] Amazon.com. Amazon EC2. http://aws.amazon.com/ec2/. Date accessed: October, 2014.

[3] Amazon.com. Amazon S3. http://aws.amazon.com/s3. Date accessed: October, 2014.

[4] Amazon.com. Amazon EC2 Service Level Agreement. http://aws.amazon.com/ec2-sla, June 2013. Date accessed: September, 2013.

[5] Amazon.com. Amazon S3 Service Level Agreement. http://aws.amazon.com/s3-sla, June 2013. Date accessed: September, 2013.

[6] Apple. BSD File Flags. https://developer.apple.com/library/mac/#documentation/FileManagement/Conceptual/FileSystemProgrammingGUide/FileSystemDetails/FileSystemDetails.html#//apple_ref/doc/uid/TP40010672-CH8-SW1, March 2012.

[7] Ross Arnold and Tim Bell. A corpus for the evaluation of lossless compression algorithms. In James A. Storer and Martin Cohn, editors, *Proceedings of the 7th Data Compression Conference*, DCC '97, pages 201–210. IEEE Computer Society, March 1997.

[8] Giuseppe Ateniese, Özgür Dagdelen, Ivan Damgard, and Daniele Venturi. Entangled Cloud Storage. Cryptology ePrint Archive, Report 2012/511, 2012. http://eprint.iacr.org/2012/511.

[9] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An Analysis of Data Corruption in the Storage Stack. *Transactions on Storage*, 4(3):8:1–8:28, November 2008.

[10] Mihir Bellare and Chanathip Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In *Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer Berlin Heidelberg, 2000.

[11] Mihir Bellare, Phillip Rogaway, and David Wagner. EAX: A Conventional Authenticated-Encryption Mode. Cryptology ePrint Archive, Report 2003/069, September 2003. http://eprint.iacr.org/2003/069.

[12] Mihir Bellare and Amit Sahai. Non-Malleable Encryption: Equivalence between Two Notions, and an Indistinguishability-based Characterization. Cryptology ePrint Archive, Report 2006/228, 2006. http://eprint.iacr.org/.

[13] Helen A. Bergen and James M. Hogan. Data Security in a Fixed-Model Arithmetic Coding Compression Algorithm. *Computers & Security*, 11(5):445–461, September 1992.

[14] Helen A. Bergen and James M. Hogan. A Chosen Plaintext Attack On An Adaptive Arithmetic Coding Compression Algorithm. *Computers & Security*, 12(2):157–167, 1993.

[15] Daniel Bernstein. The Salsa20 family of stream ciphers. In *New Stream Cipher Designs*, volume 4986 of *Lecture Notes in Computer Science*, pages 84–97. Springer, Heidelberg, 2008.

[16] Jérémy Berthomieu, Grégoire Lecerf, and Guillaume Quintin. Polynomial root finding over local rings and application to error correcting codes. *Applicable Algebra in Engineering, Communication and Computing*, 24(6):413–443, 2013.

[17] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *ACM Transactions on Storage*, 9(4):12:1–12:33, November 2013.

[18] Eli Biham and Paul C. Kocher. A Known Plaintext Attack on the PKZIP Stream Cipher. In *Fast Software Encryption '94*, volume 1008 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[19] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The Zettabyte File System. Technical report, Sun Microsystems, 2003.

[20] Kevin D. Bowers, Ari Juels, and Alina Oprea. HAIL: A High-Availability and Integrity Layer for Cloud Storage. In *Proceedings of the 16th ACM conference on Computer and Communications Security*, CCS '09, pages 187–198, New York, NY, USA, 2009. ACM.

[21] Kevin D. Bowers, Ari Juels, and Alina Oprea. Proofs of Retrievability: Theory and Implementation. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 43–54, New York, NY, USA, 2009. ACM.

[22] Michael Burrows and David J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. Technical report, DEC Systems Research Center, May 1994.

[23] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A Digital Fountain Approach to Reliable Distribution of Bulk Data. *SIGCOMM Computer Communication Review*, 28(4):56–67, October 1998.

[24] Christian Cachin and Stefano Tessaro. Asynchronous Verifiable Information Dispersal. In *Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 503–504. Springer Berlin Heidelberg, 2005.

[25] Ning Cao, Shucheng Yu, Zhenyu Yang, Wenjing Lou, and Y. Thomas Hou. LT Codes-based Secure and Reliable Cloud Storage Service. In *Proceedings of the 2012 IEEE INFOCOM*, pages 693–701. IEEE, March 2012.

[26] Nishanth Chandran, Bhavana Kanukurthi, and Rafail Ostrovsky. Locally updatable and locally decodable codes. IACR Cryptology ePrint Archive, Report 2013/520, August 2013. http://eprint.iacr.org/2013/520.

[27] Lily Chen. http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf, October 2009. NIST Special Publication (800 Series).

[28] Mahdi Cheraghchi and Venkatesan Guruswami. Non-Malleable Coding Against Bit-wise and Split-State Tampering. *CoRR*, abs/1309.1151, 2013. http://arxiv.org/abs/1309.1151.

[29] John G. Cleary, Sean A. Irvine, and Ingrid Rinsma-Melchert. On the Insecurity of Arithmetic Coding. *Computers & Security*, 14(2):167–180, 1995.

[30] John G. Cleary and Ian H. Witten. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, 32(4):396–402, April 1984.

[31] Henry Cohn and Nadia Heninger. Approximate Common Divisors via Lattices. In *Proceedings of Tenth Algorithmic Number Theory Symposium (ANTS-X)*, pages 271–293, July 2012.

[32] Reza Curtmola, Osama Khan, and Randal Burns. Robust Remote Data Checking. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability*, StorageSS '08, pages 63–68, New York, NY, USA, 2008. ACM.

[33] Arthur de Haan. Details of the Hotmail / Outlook.com outage on March 12th. http://blogs.office.com/2013/03/13/details-of-the-hotmail-outlook-com-outage-on-march-12th/, March 2013. Date accessed: October, 2014.

[34] Casey Devet, Ian Goldberg, and Nadia Heninger. Optimally robust private information retrieval. In *21st USENIX Security Symposium (USENIX Security '12)*, pages 269–283, Bellevue, WA, 2012.

[35] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[36] Danny Dolev, Cynthia Dwork, and Moni Naor. Non-Malleable Cryptography. *SIAM Journal of Computing*, 30(2):391–437, 2000.

[37] Dropbox. Yesterday's Authentication Bug. https://blog.dropbox.com/2011/06/yesterdays-authentication-bug/, June 2011. Date accessed: October, 2014.

[38] Dropbox. Security update & new features. https://blog.dropbox.com/2012/07/security-update-new-features/, July 2012. Date accessed: October, 2014.

[39] Stefan Dziembowski, Krzysztof Pietrzak, and Daniel Wichs. Non-Malleable Codes. In *Proceedings of Innovations in Computer Science*, ICS, pages 434–452, January 2010.

[40] Sebastian Faust, Pratyay Mukherjee, Daniele Venturi, and Daniel Wichs. Efficient Non-Malleable Codes and Key-Derivation for Poly-Size Tampering Circuits. IACR Cryptology ePrint Archive, Report 2013/702, 2013. http://eprint.iacr.org/2013/702.

[41] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266.

[42] Ronald A. Fisher and Frank Yates. *Statistical tables for biological, agricultural and medical research*. Oliver and Boyd, Edinburgh, 1963.

[43] International Organization for Standardization. Document management – portable document format – part 1: Pdf 1.7, July 2008. ISO 32000-1:2008.

[44] Drupal Gardens. Websites without boundaries. Easy. http://www.drupalgardens.com. Date accessed: October, 2014.

[45] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[46] Joe Giardino. Windows Azure Blob MD5 Overview. http://blogs.msdn.com/b/windowsazurestorage/archive/2011/02/18/windows-azure-blob-md5-overview.aspx, February 2011.

[47] David W. Gillman, Mojdeh Mohtashemi, and Ronald L. Rivest. On breaking a Huffman code. *IEEE Transactions on Information Theory*, 42(3):972–976, 1996.

[48] Shafi Goldwasser and Silvio Micali. Probabilistic Encryption. *Journal of Computer and System Science*, 28(2):270–299, 1984.

[49] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Practical Oblivious Storage. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, pages 13–24, 2012.

[50] Michael T. Goodrich, Roberto Tamassia, and Andrew Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivability Conference & Exposition II, 2001*, volume 2 of *Proceedings of DISCEX '01*, pages 68–82, 2001.

[51] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, pages 135–144, Washington, DC, USA, 2004. IEEE Computer Society.

[52] Google. Google App Engine: Platform as a Service. https://cloud.google.com/appengine/docs. Date accessed: October, 2014.

[53] Google. SPDY: An experimental protocol for a faster web. http://www.chromium.org/spdy/spdy-whitepaper. Accessed: January, 2014.

[54] Google. A new approach to china. http://googleblog.blogspot.com/2010/01/new-approach-to-china.html, January 2010. Date accessed: October, 2014.

[55] Parikshit Gopalan, Richard J. Lipton, and Yan Zong Ding. Error correction against computationally bounded adversaries, April 2004. Unpublished manuscript.

[56] Marco Grangetto, Enrico Magli, and Gabriella Olmo. Multimedia Selective Encryption by Means of Randomized Arithmetic Coding. *IEEE Transactions on Multimedia*, 8(5):905–917, 2006.

[57] Venkatesan Guruswami. List Decoding with Side Information. In *Proceedings of the 18th IEEE Annual Conference on Computational Complexity*, pages 300–309, July 2003.

[58] Venkatesan Guruswami and Atri Rudra. Explicit Codes Achieving List Decoding Capacity: Error-Correction With Optimal Redundancy. *IEEE Transactions on Information Theory*, 54(1):135–150, Jan 2008.

[59] Venkatesan Guruswami and Adam Smith. Codes for computationally simple channels: Explicit constructions with optimal rate. In *51st Annual IEEE Symposium on Foundations of Computer Science*, FOCS '10, pages 723–732, 2010.

[60] Shai Halevi and Phillip Rogaway. A Tweakable Enciphering Mode. In *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 482–499. Springer Berlin Heidelberg, 2003.

[61] Danny Harnik, Ronen Kat, Dmitry Sotnikov, Avishay Traeger, and Oded Margalit. To Zip or Not to Zip: Effective Resource Usage for Real-Time Compression. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 229–241, San Jose, CA, 2013. USENIX.

[62] James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Low-overhead byzantine fault-tolerant storage. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 73–86, New York, NY, USA, 2007. ACM.

[63] James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Verifying distributed erasure-coded data. In *Proceedings of the 26th annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 139–146, New York, NY, USA, 2007. ACM.

[64] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC '94, Berkeley, CA, USA, 1994. USENIX Association.

[65] Tracey Ho, Ben Leong, Ralf Koetter, Muriel Médard, Michelle Effros, and David R. Karger. Byzantine Modification Detection in Multicast Networks With Random Network Coding. *IEEE Transactions Information Theory*, 54(6):2798–2803, 2008.

[66] The IEEE and The Open Group. The open group base specifications issue 7, 2013. IEEE Std 1003.1-2013.

[67] CompuServe Inc. Graphics interchange format, July 1990. Version 89a.

[68] Lockless Inc. Low level software to optimize performance. http://locklessinc.com/. Date accessed: October, 2014.

[69] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the Security of TLS-DHE in the Standard Model. In *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 273–293. Springer Berlin Heidelberg, 2012.

[70] Sidharth Jaggi, Michael Langberg, Sachin Katti, Tracey Ho, Dina Katabi, Muriel Médard, and Michelle Effros. Resilient Network Coding in the Presence of Byzantine Adversaries. In *26th IEEE International Conference on Computer Communications (INFOCOM 2007)*, pages 616–624, May 2007.

[71] James S. Plank and Kevin M. Greenan. Jerasure: A Library in C Facilitating Erasure Coding for Storage Applications – Version 2.0. Technical Report UT-EECS-14-721, University of Tennessee, January 2014.

[72] Ari Juels and Burton S. Kaliski, Jr. PoRs: Proofs of Retrievability for Large Files. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 584–597, New York, NY, USA, 2007. ACM.

[73] Ari Juels, James Kelley, Roberto Tamassia, and Nikos Triandopoulos. Falcon Codes: Fast, Authenticated LT Codes. Cryptology ePrint Archive, Report 2014/903, October 2014. http://eprint.iacr.org/2014/903.

[74] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), September 2000. http://www.ietf.org/rfc/rfc2898.txt.

[75] Seny Kamara and Jonathan Katz. How to encrypt with a malicious random number generator. In *Fast Software Encryption*, volume 5086 of *Lecture Notes in Computer Science*, pages 303–315. Springer Berlin Heidelberg, Febuary 2008.

[76] Chris Karlof, Naveen Sastry, Yaping Li, Adrian Perrig, and J.D. Tygar. Distillation Codes and Applications to DoS Resistant Multicast Authentication. In *Proceedings of the 11th Network and Distributed Systems Security Symposium (NDSS)*, pages 37–56, 2004.

[77] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2008.

[78] Jonathan Katz and Moti Yung. Unforgeable encryption and chosen ciphertext secure modes of operation. In *Proceedings of the 7th International Workshop on Fast Software Encryption*, volume 1978 of *Lecture Notes in Computer Science*, pages 284–299, London, UK, 2001. Springer-Verlag.

[79] James Kelley and Roberto Tamassia. Secure Compression: Theory & Practice. Cryptology ePrint Archive, Report 2014/113, March 2014. http://eprint.iacr.org/2014/113.

[80] John Kelsey. Compression and Information Leakage of Plaintext. In *Revised Papers from the 9th International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 263–276, London, UK, 2002. Springer-Verlag.

[81] Hyungjin Kim, Jiangtao Wen, and J. D. Villasenor. Secure Arithmetic Coding. *IEEE Transactions on Signal Processing*, 55(5):2263–2272, 2007.

[82] Donald Knuth. *The Art of Computer Programming vol. 2: Seminumerical Algorithms*. Addison-Wesley, third edition, 1998.

[83] Ralf Koetter. Fast generalized minimum-distance decoding of algebraic-geometry and Reed-Solomon codes. *IEEE Transactions on Information Theory*, 42(3):721–737, May 1996.

[84] Ralf Koetter, Jun Ma, and Alexander Vardy. The Re-Encoding Transformation in Algebraic List-Decoding of Reed-Solomon Codes. *IEEE Transactions on Information Theory*, 57(2):633–647, February 2011.

[85] Tadayoshi Kohno. Attacking and Repairing the WinZip Encryption Scheme. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 72–81, New York, NY, USA, 2004. ACM.

[86] Tadayoshi Kohno, John Viega, and Doug Whiting. CWC: A high-performance conventional authentication encryption mode. In *Fast Software Encryption*, volume 3017 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, January 2004.

[87] Hugo Krawczyk. Distributed Fingerprints and Secure Information Dispersal. In *Proceedings of the 12th annual ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 207–218, New York, NY, USA, 1993. ACM.

[88] Hugo Krawczyk. Secret Sharing Made Short. In *Advances in Cryptology – CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 136–146. Springer Berlin Heidelberg, 1994.

[89] Hugo Krawczyk. The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?). In *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331. Springer Berlin Heidelberg, 2001.

[90] Sebastian Kreft and Gonzalo Navarro. LZ77-Like Compression with Fast Random Access. In *Proceedings of the 2010 Data Compression Conference*, DCC '10, pages 239–248, Washington, DC, USA, 2010. IEEE Computer Society.

[91] Maxwell N. Krohn, Michael J. Freedman, and David Mazières. On-the-Fly Verification of Rateless Erasure Codes for Efficient Content Distribution. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 226–240, May 2004.

[92] T. Krovetz. UMAC: Message Authentication Code using Universal Hashing. RFC 4418, March 2006. http://www.ietf.org/rfc/rfc4418.txt.

[93] T. Krovetz and W. Dai. VMAC: Message Authentication Code using Universal Hashing. http://tools.ietf.org/html/draft-krovetz-vmac-01, 2007.

[94] M. Oğuzhan Külekci. On scrambling the Burrows-Wheeler Transform to Provide Privacy in Lossless Compression. *Computers & Security*, 31(1):26–32, 2012.

[95] Michael Langberg. Private codes or succinct random codes that are (almost) perfect. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '04, pages 325–334, Washington, DC, USA, 2004. IEEE Computer Society.

[96] Shujun Li, Chengqing Li, and Jay C.-C. Kuo. On the Security of a Secure Lempel-Ziv-Welch (LZW) Algorithm. In *2011 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–5, 2011.

[97] Jen Lim, Colin Boyd, and Ed Dawson. Cryptanalysis of Adaptive Arithmetic Coding Encryption Schemes. In *Information Security and Privacy*, volume 1270 of *Lecture Notes in Computer Science*, pages 216–227. Springer Berlin Heidelberg, 1997.

[98] Richard J. Lipton. A new approach to information theory. In *11th Annual Symposium on Theoretical Aspects of Computer Science*, volume 775 of *Lecture Notes in Computer Science*, pages 699–708. Springer Berlin Heidelberg, 1994.

[99] José Lopes and Nuno Neves. Robustness of the RaptorQ FEC Code Under Malicious Attacks. In *Inforum*, May 2013.

[100] José Lopes and Nuno Neves. Stopping a Rapid Tornado with a Puff. In *Proceedings of the 2014 IEEE Conference on Security and Privacy*, S&P '14. IEEE Computer Society, 2014.

[101] M. Luby, A. Shokrollahi, M. Watson, and T. Stockhammer. Raptor Forward Error Correction Scheme for Object Delivery. http://www.ietf.org/rfc/rfc5053.txt, October 2007.

[102] M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, and L. Minder. RaptorQ Forward Error Correction Scheme for Object Delivery. http://www.ietf.org/rfc/rfc6330.txt, August 2011.

[103] Michael Luby. LT Codes. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, FOCS '02, pages 271–281, Washington, DC, USA, 2002. IEEE Computer Society.

[104] Michael G. Luby and Michael Mitzenmacher. Verification-Based Decoding for Packet-Based Low-Density Parity-Check Codes. *IEEE Transactions on Information Theory*, 51(1):120–127, 2005.

[105] Mihai Bogdan Luca, Ru Serbanescu, Stephane Azou, and Gilles Burel. A New Compression Method Using a Chaotic Symbolic Approach. In *Proceedings of the IEEE Communications Conference*, Los Alamitos, CA, USA, 2004. IEEE Computer Society Press.

[106] Anna Lysyanskaya, Roberto Tamassia, and Nikos Triandopoulos. Multicast Authentication in Fully Adversarial Networks. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 241–255, 2004.

[107] James L. Massey. Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, 15(1):122–127, January 1969.

[108] Petar Maymounkov. Online codes. Technical report, Secure Computer Systems Group, New York University, 2002.

[109] Robert J. McEliece. The Guruswami-Sudan Decoding Algorithm for Reed-Solomon Codes. *Interplanetary Network Progress Report 42-153*, pages 1–60, May 2003.

[110] David A. McGrew and John Viega. The Galois/Counter Mode of Operation (GCM). NIST Modes of Operation Process, June 2005. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-revised-spec.pdf.

[111] Silvio Micali, Chris Peikert, Madhu Sudan, and David A. Wilson. Optimal Error Correction for Computationally Bounded Noise. *IEEE Transactions on Information Theory*, 56(11):5673–5680, Nov 2010.

[112] Microsoft. What is NTFS? http://technet.microsoft.com/en-us/library/cc758691(v=ws.10).aspx, March 2003.

[113] Rich Miller. Outage in Dublin Knocks Amazon, Microsoft Data Centers Offline. http://www.datacenterknowledge.com/archives/2011/08/07/lightning-in-dublin-knocks-amazon-microsoft-data-centers-offline/, August 2011. Date accessed: October, 2014.

[114] Michael Mitzenmacher and George Varghese. Biff (Bloom Filter) Codes: Fast Error Correction for Large Data Sets. *CoRR*, abs/1208.0798, 2012.

[115] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[116] Craig G. Nevill-Manning and Ian H. Witten. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7(1):67–82, September 1997.

[117] Donald J. Newman and Lawrence Shepp. The double dixie cup problem. *The American Mathematical Monthly*, 67(1):58–61, January 1960.

[118] University of Canterbury. The Canterbury Corpus. http://corpus.canterbury.ac.nz/, November 2001. Accessed September 12, 2013.

[119] Rafail Ostrovsky, Omkant Pandey, and Amit Sahai. Private locally decodable codes. In *Automata, Languages and Programming*, volume 4596 of *Lecture Notes in Computer Science*, pages 387–398. Springer Berlin Heidelberg, 2007.

[120] Ravi Palanki and Jonathan S. Yedidia. Rateless codes on noisy channels. In *Proceedings of International Symposium on Information Theory*, ISIT '04, July 2004.

[121] Alain Pannetrat and Refik Molva. Efficient Multicast Packet Authentication. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS '03. The Internet Society, 2003.

[122] Rafael Pass, Abhi Shelat, and Vinod Vaikuntanathan. Relations Among Notions of Non-malleability for Encryption. In *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 519–535. Springer Berlin Heidelberg, 2007.

[123] Kenneth G. Paterson, Thomas Ristenpart, and Thomas Shrimpton. Tag Size Does Matter: Attacks and Proofs for the TLS Record Protocol. In *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 372–389. Springer Berlin Heidelberg, 2011.

[124] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.

[125] Juan Carlos Perez. Google Drive hit by three outages this week. http://www.networkworld.com/article/2164680/applications/google-drive-hit-by-three-outages-this-week.html, March 2013. Date accessed: October, 2014.

[126] Jonathan Perry. libwireless and wireless-python. Available at http://www.yonch.com/wireless, (2014/05/06).

[127] Jonathan Perry, Hari Balakrishnan, and Devavrat Shah. Rateless spinal codes. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 6:1–6:6, New York, NY, USA, 2011. ACM.

[128] Jonathan Perry, Peter A. Iannucci, Kermin E. Fleming, Hari Balakrishnan, and Devavrat Shah. Spinal Codes. *SIGCOMM Computer Communication Review*, 42(4):49–60, August 2012.

[129] Hossein Pishro-Nik and Faramarz Fekri. On raptor codes. In *IEEE International Conference on Communications 2006*, volume 3 of *ICC '06*, pages 1137–1141, June 2006.

[130] Raluca Ada Popa, Jacob R. Lorch, David Molnar, Helen J. Wang, and Li Zhuang. Enabling security in cloud storage SLAs with CloudProof. In *Proceedings of the 2011 USENIX Annual Technical Conference*, USENIXATC '11, pages 31–45, Berkeley, CA, USA, 2011. USENIX Association.

[131] Angelo Prado, Neal Harris, and Yoel Gluck. BREACH: REVIVING THE CRIME ATTACK. http://breachattack.com/resources/BREACH-SSL,gonein30seconds.pdf, 2013.

[132] Digital Video Broadcasting Project. Interaction channel for satellite distribution systems. Technical Report ETSI EN 301 790, European Telecommunications and Standards Institute (ETSI), May 2009. v1.5.1.

[133] Digital Video Broadcasting Project. IP Datacast over DVB-H. Technical Report ETSI TS 102 591-1, European Telecommunications and Standards Institute (ETSI), February 2010. v1.3.1.

[134] Hadoop Project. HDFS Architecture. http://hadoop.apache.org/docs/r2.5.0/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. Accessed August, 2014.

[135] Hadoop Project. HDFS RAID. http://wiki.apache.org/hadoop/HDFS-RAID. Accessed August, 2014.

[136] Qualcomm. RaptorQ Forward Error Correction Technology Overview and Use Cases. https://www.qualcomm.com/media/documents/files/raptor-technology-overview.pdf, October 2012. Qualcomm Research Presentation.

[137] Guillaume Quintin. The decoding library, version 0.4. http://www.lix.polytechnique.fr/~quintin/decoding/, August 2014.

[138] Irving S. Reed and Gustave Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, June 1960.

[139] Juliano Rizzo and Thai Duong. The CRIME Attack. http://netifera.com/research/crime/CRIME_ekoparty2012.pdf, 2012.

[140] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications. http://csrc.nist.gov/groups/ST/toolkit/rng/, April 2010.

[141] Mutsuo Saito and Makoto Matsumoto. SIMD-Oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator. In *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 607–622. Springer Berlin Heidelberg, 2008.

[142] Sumanta Sarkar and Reihaneh Safavi-Naini. Proofs of Retrievability via Fountain Code. In *Proceedings of the 5th International Conference on Foundations and Practice of Security*, FPS '12, pages 18–32, Berlin, Heidelberg, 2013. Springer-Verlag.

[143] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. *Proceedings of the VLDB Endowment*, 2013.

[144] Julian Seward. bzip2. http://www.bzip.org, 2007.

[145] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '08, pages 90–107, Berlin, Heidelberg, 2008. Springer-Verlag.

[146] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[147] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical dynamic proofs of retrievability. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 325–336, New York, NY, USA, 2013. ACM.

[148] Amin Shokrollahi. Raptor codes. *IEEE/ACM Transactions on Networking*, 14(SI):2551–2567, June 2006.

[149] Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. Venus: verification for untrusted cloud storage. In *Proceedings of the 2010 ACM workshop on Cloud Computing Security Workshop*, CCSW '10, pages 19–30, New York, NY, USA, 2010. ACM.

[150] Adam Smith. Scrambling adversarial errors using few random bits, optimal information reconciliation, and better private codes. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 395–404, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[151] David M. Smith. The Cost of Lost Data. *Graziado Business Review*, 6(3), 2003.

[152] Martin Stanek. Attacking Scrambled Burrows-Wheeler Transform. Cryptology ePrint Archive, Report 2012/149, December 2012. http://eprint.iacr.org/2012/149.

[153] Michael Stay. ZIP Attacks with Reduced Known Plaintext. In *Revised Papers from the 8th International Workshop on Fast Software Encryption*, FSE '01, pages 125–134, London, UK, UK, 2002. Springer-Verlag.

[154] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 253–267, May 2013.

[155] STMicroelectonics and INIRA. LDPC FEC Codes. http://planete-bcast.inrialpes.fr/rubrique.php3?id_rubrique=5, 2006.

[156] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. POTSHARDS: secure long-term storage without encryption. In *In Proceedings of the 2007 USENIX Annual Technical Conference*, ATC '07, Berkeley, CA, USA, 2007. USENIX Association.

[157] Roberto Tamassia and Nikos Triandopoulos. On the Cost of Authenticated Data Structures. Technical report, Brown University, 2003.

[158] Christophe Tartary and Huaxiong Wang. Rateless Codes for the Multicast Stream Authentication Problem. In *Advances in Information and Computer Security*, volume 4266 of *Lecture Notes in Computer Science*, pages 136–151. Springer Berlin Heidelberg, 2006.

[159] AWS Team. Summary of the October 22, 2012 AWS Service Event in the US-East Region. https://aws.amazon.com/message/680342/. Date accessed: October, 2014.

[160] AWS Team. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. http://aws.amazon.com/message/65648/, April 2011. Date accessed: October, 2014.

[161] AWS Team. Summary of the AWS Service Event in the US East Region. http://aws.amazon.com/message/67457/, July 2012. Date accessed: October, 2014.

[162] Cihangir Tezcan and Serge Vaudenay. On Hiding a Plaintext Length by Preencryption. In *Proceedings of the 9th International Conference on Applied Cryptography and Network Security*, ACNS '11, pages 345–358, Berlin, Heidelberg, 2011. Springer-Verlag.

[163] Emmanuel Thomé, Pierrick Gaudry, Alexander Kruppa, Guillame Quintin, Luc Sanselme, Paul Zimmermann, and Hazma Jeljeli. Mpfq decoding libary. https://gforge.inria.fr/projects/mpfq/, June 2013.

[164] Cong Wang, Qian Wang, Kui Ren, Ning Cao, and Wenjing Lou. Toward Secure and Dependable Storage Services in Cloud Computing. *IEEE Transactions on Services Computing*, 5(2):220–232, 2012.

[165] Terry A. Welch. A Technique for High-Performance Data Compression. *Computer*, 17(6):8–19, 1984.

[166] Stephen B. Wicker and Vijay K. Bhargava, editors. *Reed-Solomon Codes and Their Applications*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[167] Ian H. Witten and John G. Cleary. On the Privacy Afforded by Adaptive Text Compression. *Computer Security*, 7(4):397–408, August 1988.

[168] Kwok-Wo Wong, Qiuzhen Lin, and Jianyong Chen. Simultaneous Arithmetic Coding and Encryption Using Chaotic Maps. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 57(2):146–150, 2010.

[169] Kwok-Wo Wong and Ching-Hung Yuen. Performing Compression and Encryption Simultaneously using Chaotic Map. In *Proceedings of the First International Workshop on Nonlinear Dynamics and Synchronization*, 2008.

[170] Chung-Ping Wu and C.-C. Jay Kuo. Design of Integrated Multimedia Compression and Encryption Systems. *IEEE Transactions on Multimedia*, 7(5):828–839, 2005.

[171] Dahua Xie and C.-C. Jay Kuo. Secure Lempel-Ziv Compression with Embedded Encryption. In *Security, Steganography, and Watermarking of Multimedia Contents VII*, volume 5681 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pages 318–327, March 2005.

[172] Jiantao Zhou, Oscar C. Au, Xiaopeng Fan, and Peter Hon-Wah Wong. Secure Lempel-Ziv-Welch (LZW) Algorithm with Random Dictionary Insertion and Permutation. In *2008 IEEE International Conference on Multimedia and Expo*, pages 245–248, 2008.

[173] Jiantao Zhou, Oscar C. Au, and Peter Hon-Wah Wong. Adaptive Chosen-Ciphertext Attack on Secure Arithmetic Coding. *IEEE Transactions on Signal Processing*, 57(5):1825–1838, 2009.

[174] Jacob Ziv and Abraham Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978.