Secure Compression: Theory & Practice

James Kelley and Roberto Tamassia
Dept. of Computer Science, Brown University
{jakelley,rt}@cs.brown.edu

March 12, 2014

Abstract

Encryption and compression are frequently used together in both network and storage systems, for example in TLS. Despite often being used together, there has not been a formal framework for analyzing these combined systems; moreover, the systems are usually just a simple chaining of compression followed by encryption. In this work, we present the first formal framework for proving security in combined compression-encryption schemes and relate it to the traditional notion of semantic security. We call this *entropy-restricted semantic security.* Additionally, we present a new, efficient cipher, called the squeeze cipher, that combines compression and encryption into a single primitive and provably achieves our entropy-restricted security.

Keywords. Compression, Encryption, Secure compression, Entropy, Provable security

1 Introduction

Individuals and organizations are producing and retaining data in volumes that are unprecedented. One technique widely used to reduce the footprint of the data is to simply filter it through a compression algorithm before trasmission or storage. Indeed, many popular file systems utilize transparent file compression, including Microsoft's NTFS [23], Apple's HFS+ [1], and Oracle's ZFS [26]; additionally, both the standard internet protocol HTTP [9] and Google's own SPDY [12] support compression. Concurrent with the accumulation of mountains of data, thefts of the data have been increasing in frequency, sophistication, and aggression. Encryption plays a key role in securing the data from these attacks both while at rest (e.g., in NTFS, HFS+, and ZFS) and while in transit (e.g., TLS [8] 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.

Unfortunately, there are some drawbacks to combining compression and encryption together in a naïve fashion. Firstly, without the creation of special constructions merging compression and encryption, the system must perform two passes over the data: compressing then encrypting. It would be ideal to achieve a compressed and encrypted output in a single pass over the data. While the compression and encryption could be set up to operate in parallel (i.e., data is encrypted in a streaming manner as it comes out of the compression routine), there is still "twice" the amount of work being done. Though it seems that current systems employing compression and encryption are not achieving optimal efficiency, more troubling, by combining these primitives together it becomes impossible to achieve semantic security. Specifically, in the standard security game, the advesary $\mathcal{A}$ is permitted to select *any* two messages to be encrypted; thus, $\mathcal{A}$ may simply select one to consist entirely of 0's and select the other uniformly at random. With overwhelming probability, these will compress to different lengths and hence produce ciphertexts of different lengths, making them trivially distinguishable. In addition to this, it has been observed in [16] that the compressibility of a file is a side-channel leaks non-trivial information about the file itself, giving hints about both the content and the interal structure of the file. See [16] for a full discussion of the information leakage of a general composition of compression and encryption and [27] for a real-world exploit using this side-channel.

Contributions. Our contributions to this topic are several. First, in this work we present the first formal definition of security for combined compression-encryption systems, generalizing semantic security in a natural way to apply to compressing ciphers. Futhermore, this definition readily extends to provide definitions for both adaptive and non-adaptive chosen-ciphertext security. In addition, we provide a novel construction for a compressing cipher that provably achieves our new security definition and is both theoretically and practically efficient.

Organization. The remainder of this paper is organized as follows. In Section 2, we provide preliminary definitions for the tools that will be used throughout this work. Section 3 details our construction for secure compression called the squeeze cipher. Section 4 provides a detailed security analysis of the squeeze cipher proving that it achieves the claimed level of security. Section 5 provides a brief experimental evaluation of the cipher showing its practicality. And finally, before we conclude, Section 6 provides a high-level overview of previous work on secure compression.

2 Preliminaries

Notation. We denote the security parameter by λ and the empty string by Λ. Let PPT stand 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 $x \overset{R}{\leftarrow} S$ denote sampling x from the set S uniformly at random. 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 $\{0,1\}^n$.

2.1 Basic Definitions

First, we provide a definition of data compression. Roughly, data compression is an encoding of a sample from a data source such that, on average, an encoded message is no longer than the original data. We concern ourselves with being shorter on average as it is not possible to achieve compression in all cases since, for example, there are 2^k k-bit binary strings and only $2^k - 1$ strings of length less than k.

Definition 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 $C : \mathcal{M} \to \{0,1\}^*$. 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.* $\square$

We only consider uniquely decodable codes. Let $p(a)$ be the probability of sampling a from $\mathcal{M}$ according to $\mathcal{D}$. The *average length* of a code is defined as, $L(C) = \sum_{m\in\mathcal{M}} p(m)l(C(m))$. Hopefully, we have that $L(C) \leq \sum_{m\in\mathcal{M}} p(m)l(m)$. If this holds, then we will just 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 will call D the *decompression function.*

We define a *keyed compression function* to be a compression function that takes an additional parameter k, such that k is required to correctly decompress the input. Any plain compression scheme trivially satisfies this definition by having the key-generation algorithm $\mathcal{K}$ always output $\perp$ and by letting $\mathcal{C}$ and $\mathcal{D}$ 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 also satisfies this definition since the identity function is a valid compression function.

Definition 2. A *keyed compression function* over a message space $\mathcal{M} \subseteq A^*$ is a triple of (possibly probabilistic) polynomial-time algorithms $(\mathcal{K}, \mathcal{C}, \mathcal{D})$ where,

- $\mathcal{K}$ takes as input 1^λ and outputs a key k.
- $\mathcal{C}$ is a compression function that takes as input a key k and a plaintext $m \in \mathcal{M}$ and outputs a (compressed) ciphertext $c \in \{0,1\}^*$.
- $\mathcal{D}$ takes as input a key k and a ciphertext $c \in \{0,1\}^*$ and outputs the plaintext $m \in \mathcal{M} \cup \{\perp\}$.

We require that for all $k \in [\mathcal{K}(1^\lambda)]$, for all $m \in \mathcal{M}$, $\mathcal{D}(k, \mathcal{C}(k,m)) = m$. □

As part of the construction of our compressing cipher, we will use a *pseudo-random permutation* (PRP) as part of 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 $k \leftarrow \{0,1\}^\lambda$). We do not give the adverary $\mathcal{A}$ 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 3. A keyed permutation $f : \{0,1\}^\lambda \times \{0,1\}^n \rightarrow \{0,1\}^n$ is (t, q, ε)*-indistinguishable*, if for all PPT distinguishers $\mathcal{D}$ running in time t and making q queries to $\mathcal{O}_f$, successfully distinguishes f from a permutation p chosen uniformly at random from $\mathcal{P}(n)$ 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.*

We also utilize a pseudo-random generator G in our compressing cipher to help randomize the compression and decompression functions. Since we would like our scheme to be able to encrypt multiple messages, we require the PRG to have 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 pseudo-random. 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.[1] A PRG that is secure in this setting we call *reseedably-indistinguishable.*

Definition 4. 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 be reseedably-indistinguishable does not significantly change its strength. In Appendix A we formally prove the polynomial-equivalence of

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

reseedable-indistinguishability and standard indistinguishability through a straightforward hybrid argument. We summarize this result with the following lemma.

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

2.2 Entropy-restricted Semantic Security

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. 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.

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 $\mathcal{C} \subset \mathcal{M}$. 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—where two messages are equivalent if their encryptions are indistinguishable. 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 adding random padding until all ciphertexts have length l_2.

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 $m_0, m_1 \in \mathcal{C}$, for some class $\mathcal{C} \subset \mathcal{M}$. 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.

Definition 5. We say that a symmetric cryptosystem $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is (t, q, ε)-*entropy-restricted CPA secure* for a class of messages $\mathcal{C} \subset \mathcal{M}$, 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. $\square$

If t and q are both polynomially bounded, and ε is negligible, then we simply say that Π is *ER-CPA secure.* Stronger definitions of security are possible including analogous definitions for entropy-restricted versions of IND-CCA-1 and IND-CCA-2 security (called ER-CCA-1 and ER-CCA-2, respectively). We will use only ER-CPA security in this work. 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 $l_1 < l_2$. Note that any IND-CPA secure cryptosystem combined with a compression pre-processing step is automatically ER-CPA secure. 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. 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

Algorithm 1 The LZW compression algorithm.

```
Input: Character stream $I$
Output: Stream of table indices
  Initialize table $T$ to contain all single-character strings.
  $prefix \leftarrow \Lambda$                                ▷ The current prefix of the input
  $i \leftarrow$ number of single-character strings           ▷ The index of the first free entry in $T$
  while there is more input in $I$ do
      Read next character $c$
      if $prefix \circ c$ in $T$ then
          $prefix \leftarrow prefix \circ c$             ▷ Extend prefix with $c$ and continue processing input
      else
          Output index of $prefix$ in $T$                   ▷ $prefix \circ c$ not in $T$: add it to $T$
          $T[i] \leftarrow prefix \circ c$
          $i \leftarrow i + 1$
          $prefix \leftarrow c$                      ▷ Continuing processing the input starting at $c$
  if $prefix \neq \Lambda$ then                             ▷ Output index of the remaining input
      Output index of $prefix$ in $T$
```

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 [15] also hold in our entropy-restricted context.

2.3 LZW Compression

In [31], Welch proposed a modification of the older LZ78 compression algorithm [38]. The proposed algorithm, known as LZW, is a dictionary-based compression algorithm where the dictionary is constructed as the input is processed. The dictionary D starts with all single-character strings. At each iteration, the next characters are scanned and the algorithm matches the longest prefix of the input p 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 the process on the remaining input starting with the next character c. This is shown in Algorithm 1.

Decompression works in much the same way but process indices instead of characters: reading in the next index, looking up the corresponding entry in D, and outputting the string s. It then takes the first character c of s and appends it to the previous output string s', inserting $s' \circ c$ into D. Decompressing is detailed in Algorithm 4 in Appendix B. Note that there is a special case in decompression: the input may contain the string $c \circ p \circ c \circ p \circ c$, where $c \circ p$ is already in D. The compression algorithm will match $c \circ p$ and insert $c \circ p \circ c$ into D, then match $c \circ p \circ c$ and output its index. But, at the receiver, the second index refers to an *empty* entry in D. Since we assume no errors in the input, we know that if the index points to the next available cell (where the new entry will be placed), then we are in 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 table 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 table at the same time.

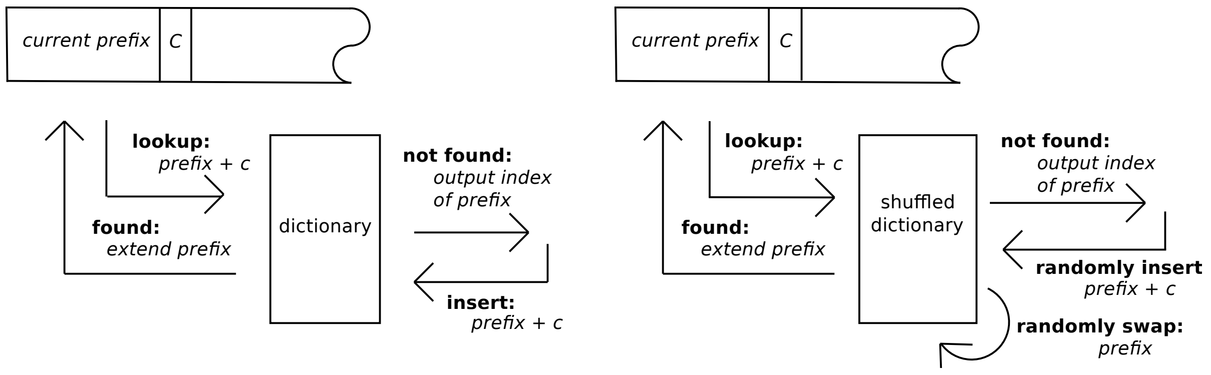

Figure 1: LZW compression (left) compared with the squeeze cipher (right).

Typically, the table is doubled in size (and outputs become one bit longer) when an entry fills the table. This approach is most common as it gives better compression, using fewer bits early on in the encoding process. 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).

3 Squeeze Cipher

To turn the LZW algorithm into a cipher, we change the management of the dictionary D and randomize it. Briefly, we randomize the layout of D using a PRG and then incrementally re-randomize it while encrypting the input message (to ensure that we do not simply have a fancy substitution cipher). As a first attempt, we could simply randomly permute D after each iteration. This guarantees that each dictionary entry's index is uniformly distributed, and the cipher easily achieves perfect secrecy. However, this is quite inefficient as it requires $O(n \log n)$ bits and $O(n)$ time to generate and apply a random permutation to the dictionary.[2] Thus, we would have at least a factor of n slowdown in the speed of the cipher relative to the plain LZW algorithm.

To avoid this gross inefficiency, we only randomize the entry that was used and re-use the remaining "unused" randomness. That is, each time an entry in D is used, it is swapped with another (possibly empty) entry in D chosen at random. This leaves much of D untouched and the unused randomness (i.e., the random positions of the other entries) can be re-used. When a new entry is inserted, its position is chosen at random from the *unoccupied* cells (accomplished by repeatedly sampling an index at random until an empty cell is encountered). Note that if the dictionary is a random permutation of k out n elements, then selecting at random from the empty cells produces a random $k+1$ out of n permutation.

The number of times this sampling must be repeated depends on the load in the table (i.e., the ratio of occupied cells to the total number of cells)—denote the load in D by α. To keep the number of samples small, with high-probability, we put an upper bound on α, call this β. With high-probability, there will be at most a logarithmic number of samples, but on average, there will be a only constant number (i.e., $1/(1-\beta)$). Note, however, that by leaving part of D empty, we sacrifice some compressibility to achieve better speed. LZW, on the other hand, has no such requirement and may use every entry in D.

At the start of encryption, an initialization vector iv is generated (e.g., via a counter) and used to help seed the PRG G. Specifically, we apply a pseudo-random permutation $f : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$, where λ is the length of the IV, and compute the seed by $f_k(iv) = seed$, where k is the secret

[2]The Fisher-Yates algorithm can generate a permutation of n elements in $O(n)$ time [10].

Algorithm 2 Squeeze cipher encryption.

```
Input: key $k_{\text{prp}}$, input character stream $I$
Output: encrypted output stream $O$
  InitSqueeze($k_{\text{prp}}$, $iv$, $G$, $prefix$, $c$, $T$)
  while there is more input in $I$ do          ▷ Start compressing and encrypting stream $I$
      Read next character $c$
      if $prefix \circ c$ in $T$ then
          $prefix \leftarrow prefix \circ c$
      else
          Output index $i$ of $prefix$ in $T$
          RandomSwap($i$, $T$)                  ▷ swap $prefix$ with a random entry in $T$
          RandomInsert($prefix \circ c$, $T$)    ▷ insert $prefix \circ c$ randomly into $T$
          $prefix \leftarrow c$
  if $prefix \neq \Lambda$ then                 ▷ Make sure we get any straggling input
      Output index $i$ of $prefix$ in $T$
```

key. The IV is prepended to the output so that receiver can properly initialize G for decryption. Note that since f is a PRP, the output of G seeded with *seed* is indistinguishable from its output when seeded with a truly random seed.

The decryption algorithm is specified in Algorithm 5 located in Appendix C. It behaves much as the original decompression algorithm in Algorithm 4 but with one subtle complication. Since we are randomly inserting entries into D, we have to be more careful with the special case described earlier, where $prev = c \circ s$ is sent followed by $prev \circ c = c \circ s \circ c$.[3] Here, instead of testing that the received index k is equal to the next free index, we peek ahead at the pseudo-random index for the next inserted entry (denoted r) and check if $k = r$. Note that this is *not* simply a matter of looking at the next few pseudo-random bits. Since there can be a sequence of collisions during insertion and the inserted entry may be evicted multiple times, we must find the final resting place of the inserted entry by pre-computing the entire the random insertion procedure.

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 squeeze, the compression and decompression functions can also use these efficient data structures for look-ups and processing the input/output characters. The primary different between squeeze and LZW is the dictionary management where squeeze partially randomizes the dictionary at each step with RandomSwap and RandomInsert. Assume that it takes $O(1)$ time for the PRG to produce a random index. Then, the RandomSwap step only takes $O(1)$ time, but RandomInsert could take more. Note that since the load in the dictionary is $\alpha \leq \beta$ for some bound β, and since each iteration of the while-loop is independent of the others, on average there will be $1/(1-\beta)$ iterations. Moreover, with high-probability there will be at most $\log_{1/(1-\beta)} d$ iterations, where d is the size of the dictionary (note that $d < n$). Thus, we have that both encryption and decryption take $O(n \log d)$ time with high-probability.

[3]Recall that, technically, we receive an index for an entry that is not yet defined (but will be next).

Algorithm 3 The InitSqueeze and RandomInsert functions.

```
function InitSqueeze($k_{\text{prp}}$, $iv$, $G$, $p$, $c$, $T$)
    Generate fresh $iv$
    Compute $seed = f_{k_{\text{prp}}}(iv)$
    Initialize $G$ with $seed$
    $p \leftarrow \Lambda$, $c \leftarrow \Lambda$
    Initialize table $T$
    for each single character string $s$ do
        RandomInsert($s$, $T$)

function RandomInsert($s$, $T$)
    Choose a random index $r$ in $T$
    while $T[r] \neq \Lambda$ do
        Choose a new random index $r$ in $T$
    $T[r] \leftarrow s$

function RandomSwap($i$, $T$)
    Choose a random index $r$ in $T$
    Swap $T[i]$ and $T[r]$
```

4 Security

In this section we prove the ER-CPA security of the squeeze cipher. We proceed by first proving two lemmas that establish two loop-invariants for the encryption and decryption; then we will prove the main theorem that squeeze is ER-CPA secure. The security of squeeze hinges on the random permutation of the dictionary entries. Intuitively, if the entries are randomly permuted, then an adversary observing the output cannot derive any correlations between the output indices and the input strings. In our construction, we are outputting a sequence of dictionary entries and in so doing, we provide glimpses into the internal ordering of the entries in the dictionary. The goal of each of RandomSwap and RandomInsert is to restore the random permutation.

Lemma 1. *Given a random k permutation of n elements, if we reveal an arbitrary entry, applying* RandomSwap *to that entry produces a random permutation of k out of n elements.*

Proof. Given a k out of n permutation, there are $n!/(n-k)!$ possible permutations. After revealing one entry, we now have a $k-1$ out of $n-1$ permutation, giving $\frac{(n-1)!}{((n-1)-(k-1))!} = \frac{(n-1)!}{(n-k)!}$ possible permutations. RandomSwap selects an element uniformly 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. Note that the $k-1$ out of $n-1$ permutation is uniformly distributed, and since the swap is uniformly distributed among n possibilities, the final k out of n permutation is uniformly distributed. □

Thus, RandomSwap restores the 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. *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 $k+1$ permtuations 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 encoder or decoder, the positions of the dictionary's entries form a random k out of n permutation. Now we prove that the squeeze cipher is ER-CPA secure. Squeeze is, at its 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. Moreover, this reduction is tight. In the proof, for simplicity, we assume all encryption oracle queries take time at most t_e. Let $\mathcal{C}(G, f)$ denote squeeze instantiated with PRG G and PRP f.

Theorem. *Let G be a (t_1, r, ε_1)-reseedably-indistinguishable PRG that takes seeds of length λ, and let f be a (t_2, r, ε_2)-indistinguishable PRP over $\{0,1\}^\lambda$. Then the squeeze cipher $\mathcal{C}(G, f)$ is $(t', r-1, \delta)$-ER-CPA secure where $t' = \min\{t_1 - rt_e, t_2 - rt_e\}$ and $\delta = 2(\varepsilon_1 + \varepsilon_2) + \frac{r(r-1)}{2^{\lambda+1}}$.*

Proof. Suppose $\mathcal{A}$ can break the ER-CPA security of 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 *iv*, and then encrypts m following the specification of the squeeze cipher. 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', $\mathcal{D}$ outputs 1 if $b' = b$, and 0 otherwise.

Note that we do not use an initialization vector for generating the seed for the PRG in $\mathcal{O}$ since the seed is chosen uniformly at random and, thus, is distributed identically to $f(iv)$, when using a fresh *iv* for each encryption. 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 pseudo-random, 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_1)-indistinguishable, we have that $\delta \leq 2\varepsilon_1$, there were at most $r - 1$ reseedings, and the running time of $\mathcal{A}$ is $t' = t_1 - qt_e$.

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 squeeze is at most $\mu = r(r-1)/2^{\lambda+1}$, where r is the number of reseeding requests. Replace f, now, 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 pseudo-random, then it outputs 1, otherwise it outputs 0. $\mathcal{F}$ runs $\mathcal{A}$, and for each query m, $\mathcal{F}$ generates a fresh *iv*, queries $\mathcal{O}_f(iv) = s$, and then encrypts m in the normal way using the PRG G. 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 $\frac{1}{2} + \mu + 2\varepsilon_1$, which implies 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 $\frac{1}{2} + \mu + 2\varepsilon_1$, for some δ. This gives a total success probability of $\frac{1}{2}(\frac{1}{2} - \mu - 2\varepsilon_1) + \frac{1}{2}(\frac{1}{2} + \mu + 2\varepsilon_1 + \delta) = \frac{1}{2} + \frac{\delta}{2}$. Since f is a (t_2, r, ε_2)-indistinguishable, we know that $\delta/2 \leq \varepsilon_2$. 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 $2(\varepsilon_1 + \varepsilon_2) + \mu$. Note that the number of encryption oracle queries is exactly equal to the number of reseedings of G, and is thus upper-bounded by r. Finally, we have that $\mathcal{F}$'s running time is upper-bounded by t_2, so $\mathcal{A}$'s running time is upper-bounded by $t_2 - rt_e$. □

Corollary. *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 $\mathcal{C}(G, f)$ is ER-CPA secure.*

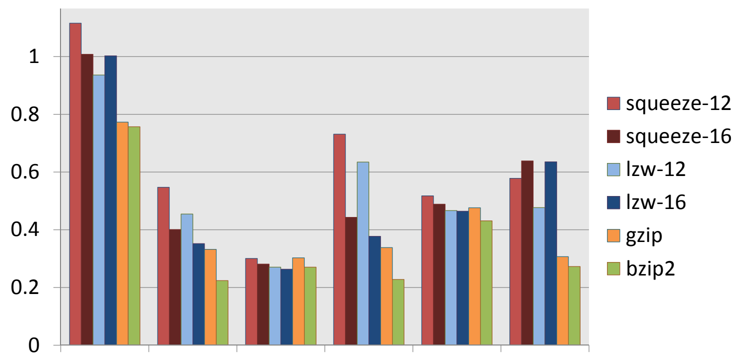

Figure 2: Chart comparing squeeze with 12-bit and 16-bit indices to standard compression algorithms. Lower numbers indicate better compression.

Attacks on Squeeze. The proof of security for squeeze given above is a straightforward reduction to the security of the PRG and the PRP. Note however, that this theorem guarantees the privacy of the scheme, but says nothing about its resilience to more powerful attacks or misuse of the cipher. The security of squeeze hinges upon the uniqueness of the initialization vector iv. Reuse of an IV will cause two (or more) ciphertexts to have the same dictionary randomization and would be vulnerable to known-plaintext attacks. Indeed, repeated use of an IV turns squeeze into a fancy substitution cipher. Correlations among the IVs are allowed since the PRP will send them to different, pseudo-random outputs. However, if the PRP is weak, correlated IVs may have correlated outputs and weaken the security of the PRG.[4]

Squeeze is also vulnerable to chosen-ciphertext attacks. In particular, given the encryption c of a challenge message m_b, an attacker $\mathcal{A}$ can flip bits in c to produce another ciphertext c'. Since the dictionary is laid out randomly with load α, any manipulated portion of the ciphertext has a probability $\approx \alpha$ of begin valid. As long as α is non-negligible, c' is likely a valid (and unique) ciphertext that may be given to a decryption oracle. If c' is produced by manipulating the tail of c, then the uncorrupted portion of c', once decrypted, will share a prefix with m_b. This is foiled if α is negligible (e.g., $< 2^{-128}$), since any corruption will almost certainly result in an invalid index and a decryption failure. But then the table is almost completely empty and the compression ratio will suffer greatly; indeed, all but the largest of inputs will certainly expand.

5 Experiments

In this section we provide a summary of experiments evaluating the compressibility and speed of the squeeze cipher as compared to LZW and the standard algorithms gzip and bzip2. The test data used is the Canterbury corpus from [25]. It is a standard corpus of documents for testing lossless compression algorithms and includes many different test files. We compare the algorithms on a subset of the corpus: a text file of random digits, the Bible, a segment of E.coli DNA, a snapshot of the CIA World Fact Book, the first million digits of π, and a C source file. These files highlight the varying performance of the algorithms on different inputs. The tests were performed in a on a 2.6GHz, Intel Core i5 processor with 4GB of RAM running Arch Linux. 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 as the PRG [3]. In each, the load is at most 0.5,

[4]PRG security guarantees assume that the seeds are independent and random, so correlated seeds may produce weakened streams. See [14] for an analysis of this problem and constructions secure against malicious inputs.

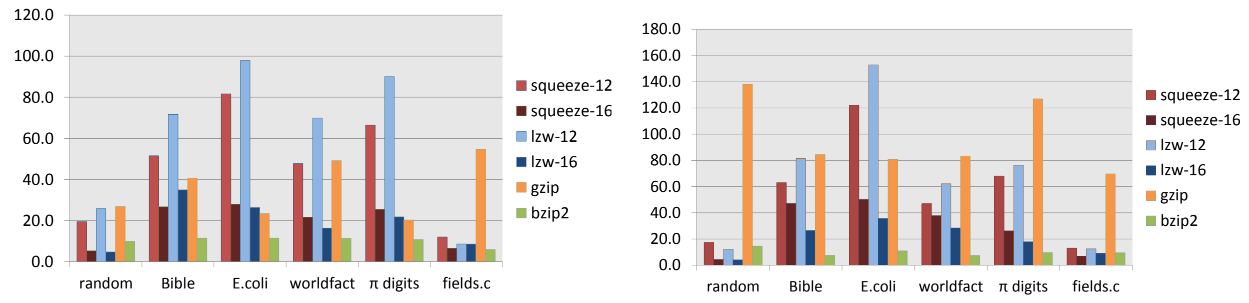

Figure 3: Charts comparing the speed of compression (left) and decompression (right) in squeeze to standard compression algorithms and LZW. Numbers are in megabytes per second.

and once the dictionary is full no entries are added or removed. Each of gzip and bzip2 are run at their default compression levels and the executables were compiled with `gcc` with the -O2 flag.

Compression Ratio. The compression ratio is defined as the (compressed) output size divided by the (uncompressed) input size. A smaller number signifies better compression. We can see in Figure 2 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, squeeze does not perform well since code does not exhibit as much regularity as, say, English text.

Compression Speed. In Figure 3, we compare the compression and decompression speeds of squeeze with those of gzip, bzip2, and lzw. Again we compare both 12-bit and 16-bit indices for squeeze and lzw, and use gzip and bzip2 at their default compression levels. The load of the table in squeeze was kept to $\frac{1}{2}$, but lzw was given a maximum load of 1 since there was no need to keep its table partially empty. Each of lzw, gzip, and bzip2 perform compression followed by AES in CBC mode. The second three files are the "large" files from the Canterbury corpus, allowing us to measure the steady-state of the algorithms. Each value in the table is the average of 10 runs.

In general, both squeeze and lzw greatly out-perform bzip2 in both compression and decompression (with one exception). However, their performance compared to gzip varies greatly. In many cases gzip has the fastest decompression speed, except for the E.coli input file—which is clearly where lzw and squeeze perform the best. The high speeds achieved by lzw and squeeze are due to the simple dictionary management scheme. Namely, they both freeze their dictionary when it is full, and can thereby boost the speed of the inner loops. This is further evidenced by the fact that gzip greatly out-performs both algorithms on the small file `fields.c`. Note that this also explains why squeeze is able to out-perform lzw on files such as `fields.c`: namely, its dictionary fills up earlier. Overall, however, we can see that squeeze is, indeed, a practical keyed compression algorithm, capable of achieving speeds comparable to standard algorithms with mature implementations.

6 Previous Work

Compression has featured as an important part of many schemes, being used as a pre-processing step before encryption. Several popular compression schemes include encryption as a post-processing step, including WinRAR, WinZIP, and PKZip. WinRAR uses AES-128, while WinZIP uses a combination of AES-128/256 and HMAC-SHA-1, but has several weaknesses detailed in [18]. PKZip used a custom stream cipher that was broken in [4] (the attack was improved in [30]). Looking at the security implications of compression, in [16] the author points out that adding compression as a pre-processing step for encryption can weaken security. For example, when using an adaptive compression algorithm, if an adversary knows a prefix of the input (e.g., packet headers), he can use the overall compressibility to learn partial information about the unknown portion. This was later demonstrated in an attack on TLS that can steal HTTP session cookies [27].

Huffman Codes. In [11], the authors 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 partial recovery of input file, and, indeed, it appears that many files can be partially (but not necessarily fully) decoded. In [34], the authors use Huffman encoding with multiple encoding trees, where an optimal tree is generated and then mutated to produce different trees. They use a secret, random ordering of the trees to encode the input symbols.

Burrows-Wheeler Transform. The Burrows-Wheeler transform, first described in [5], is an invertible partial-sorting algorithm that permutes the input characters so that identical characters are (roughly) grouped together. The authors then utilize move-to-front encoding followed by Huffman encoding to compress the data (`bzip2` is the most popular implementation [28]). In [19] the author modifies the Burrows-Wheeler transform by using a secret, randomly chosen ordering of the input alphabet. The move-to-front encoding uses a separate, secret ordering of the alphabet. This algorithm was shown to be weak in [29], succumbing to both chosen and known plaintext attacks.

LZ-based Codes. In [35] the authors propose a secure compression scheme based on the LZW algorithm similar to this work. They start with the initial dictionary randomly permuted and each new entry is inserted at a random position. There are several drawbacks to this scheme. First, they deal with collisions during insertion 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 can evict one of the single-character strings from the dictionary and cause encoding to fail. Second, their construction does not re-randomize an entry after using it. So, as long as an entry is never in a collision, the corresponding string *always* maps to the same output index, turning the scheme into a substitution cipher.

In [36], the authors propose another secure LZW algorithm using a randomized dictionary. They start by randomly permuting the initial dictionary and use a keyed hash function to insert subsequent entries into a random position. 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 is implicitly assumed that there are no collisions. Secondly, their scheme is extremely vulnerable to chosen plaintext attacks as detailed in [20]. Finally, the permutation step the apply after each encoding step is quite expensive as it touches the entire dictionary, reducing the efficiency of the algorithm from $O(n)$ to $O(nd)$ (where d is the size of the dictionary). In our construction, each encoding step is on average constant time, and encoding and decoding take time $O(n \log d)$ with high probability.

Arithmetic Coding. Arithmetic coding works by dividing the interval $[0, 1)$ into disjoint segments whose lengths correspond to the probability of a particular symbol occurring in the input stream. The algorithm reads an input symbol, sets the interval bounds to the end points of the symbol's associated interval and then recurses on that interval. At the end of the input stream, the algorithm has an interval that uniquely determines the sequence of input symbols and it outputs the shortest element in that interval. The probability distribution, called the *model*, can be fixed or dynamic.

There have been several efforts to combine arithmetic coding with encryption. One example is [17] where the authors split each interval into smaller segments and then permute them all together. The work [37] provides a chosen ciphertext attack on this scheme that breaks output of the encoder in time linear in the number of ciphertext queries. In [6], the authors present known and chosen plaintext attacks against a simple arithmetic coder with a fixed and secret input model; using $b + 2$ chosen plaintext characters, they can recover the distribution with b bits of precision.

In [32], the authors present two schemes for secure adaptive arithmetic coding: in the first, the key is the initial (random) model of encoder, while in the second the encoder starts with a fixed model but ingests a random string before encoding any input. In [2] the authors give a chosen plaintext attack on the first scheme that "floods" the encoder to force it into a known (and decodable) state. In [21], the authors build on [2] giving key-recovery attacks on both schemes, via adaptive chosen plaintext attacks. In [13], the authors propose a randomized encoder where the partitions of the interval are randomly permuted at each step. Their scheme succumbs to a chosen plaintext attack where the attacker probes the encoder with incrementally longer plaintexts to learn the ordering of the intervals at each step. The scheme in [33] divides each symbol's interval into small subintervals, permutes them all together, and then encodes via a reverse application of the skew tent map (shown to be equivalent to arithmetic coding in [22]).

7 Conclusion & Future Work

In this paper we have presented the first formal framework for analyzing combined compression and encryption schemes and give an efficient construction that is provably secure in the framework. There are several open questions with this work. First and foremost, the LZW algorithm is inherently sequential in nature (and hence, so is squeeze). Developing a *random access* variant of the squeeze ciphers would be synergistic with real-world uses of data. There exist both ciphers and compression functions that allow random access, so there does not appear to be anything inherently precluding a compressing cipher from also having this property. Moreover, allowing random access implies parallelizability (at least in decompression), which would be a boon for use on multi-core systems. Another question is whether or not these techniques can be adapted to alternative compression paradigms. Here, we provided a construction based on a self-constructing dictionary, alternatives include prediction by partial matching (e.g., [7]), grammar-based codes (e.g., [24]), and integrating a variant of the Burrows-Wheeler transform. And of course, additional optimizations in the compression and efficiency of the squeeze cipher are desirable.

Acknowledgments

Research supported in part by the National Science Foundation under grants CNS–1012060 and CNS–1228485, and by the NetApp Faculty Fellowship. We thank James Lentini, Peter Shah, and Shiva Chaitanya of NetApp for useful discussions and feedback on the squeeze cipher.

References

[1] 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.

[2] Helen A. Bergen and James M. Hogan. A Chosen Plaintext Attack On An Adaptive Arithmetic Coding Compression Algorithm. *Computers and Security*, 12:157–167, 1993.

[3] Daniel Bernstein. The salsa20 family of stream ciphers. In Matthew Robshaw and Olivier Billet, editors, *New Stream Cipher Designs*, volume 4986 of *Lecture Notes in Computer Science*, pages 84–97. Springer, Heidelberg, 2008.

[4] 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.

[5] M. Burrows and D.J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. Technical report, DEC Systems Research Center, May 1994.

[6] John G. Cleary, Sean A. Irvine, and Ingrid Rinsma-Melchert. On the Insecurity of Arithmetic Coding. *Computers & Security*, 14(2):167–180, 1995.

[7] 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.

[8] 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.

[9] 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.

[10] Ronald A. Fisher and Frank Yates. *Statistical tables for biological, agricultural and medical research*. Oliver and Boyd, Edinburgh, 1963.

[11] David W. Gillman, Mojdeh Mohtashemi, and Ronald L. Rivest. On breaking a Huffman code. *IEEE Transactions on Information Theory*, 42(3):972–976, 1996.

[12] Google. SPDY: An experimental protocol for a faster web. `http://www.chromium.org/spdy/spdy-whitepaper`. Accessed: January, 2014.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Hyungjin Kim, Jiangtao Wen, and J.D. Villasenor. Secure Arithmetic Coding. *IEEE Transactions on Signal Processing*, 55(5):2263–2272, 2007.

[18] 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.

[19] M. Oğuzhan Külekci. On scrambling the Burrows-Wheeler Transform to Provide Privacy in Lossless Compression. *Computers & Security*, 31(1):26–32, 2012.

[20] 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.

[21] 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.

[22] 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.

[23] Microsoft. What is NTFS? `http://technet.microsoft.com/en-us/library/cc758691(v=ws.10).aspx`, March 2003.

[24] 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.

[25] University of Canterbury. The Canterbury Corpus. `http://corpus.canterbury.ac.nz/`, November 2001. Accessed September 12, 2013.

[26] Oracle. Oracle Solaris ZFS File System (Introduction). `http://docs.oracle.com/cd/E26502_01/html/E29007/zfsover-1.html`, November 2012.

[27] Juliano Rizzo and Thai Duong. The CRIME Attack. `http://netifera.com/research/crime/CRIME_ekoparty2012.pdf`, 2012.

[28] Julian Seward. bzip2. `http://www.bzip.org`, 2007.

[29] Martin Stanek. Attacking Scrambled Burrows-Wheeler Transform. Cryptology ePrint Archive, Report 2012/149, December 2012. `http://eprint.iacr.org/2012/149`.

[30] 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.

[31] Terry A. Welch. A Technique for High-Performance Data Compression. *Computer*, 17(6):8–19, 1984.

[32] Ian H. Witten and John G. Cleary. On the Privacy Afforded by Adaptive Text Compression. *Computer Security*, 7(4):397–408, August 1988.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] Jacob Ziv and Abraham Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978.

A Equivalence of Indistinguishability and Reseedable Indistinguishability

Here we prove the equivalence of normal indistinguishability for a pseudo-random 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)**.** *Let G be a (t, b, ε)-indistinguishable PRG, then G is $(\varepsilon r^2, r+1, b, t-r)$-reseedably-indistinguishable.*

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 be 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 $b \overset{R}{\leftarrow} \{0,1\}$, $D_0 = U_n$ 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 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.[5] □

[5]Note that we assume that generating a random string and generating $G(s)$ for a random seed both take constant time.

B LZW Decompression

Algorithm 4 The LZW decompression algorithm.

```
Input: Sequence of indices $I$
Output: Stream of characters or the error symbol $\perp$
  Initialize table $T$ to contain all single-character strings.
  $c \leftarrow \Lambda$                              ▷ first character of next output string, initialize to empty
  $prev \leftarrow \Lambda$                           ▷ the previously output string, initialize to empty
  $i \leftarrow$ number of single-character strings
  while there is more input in $I$ do
      Read next index $k$
      if $T[k]$ is defined then                       ▷ $k$ is a valid index
          $c \leftarrow \textsc{head}(T[k])$          ▷ Take the first character of $T[k]$.
          $T[i] \leftarrow prev \circ c$              ▷ $prev \circ c$ was the value inserted after encoding $prev$.
          $i \leftarrow i + 1$
      else if $T[k]$ is not defined and $k = i$ then  ▷ Special case: original input was $prev \circ prev \circ c$
          $c \leftarrow \textsc{head}(prev)$          ▷ The first character of $prev$ is $c$.
          $T[k] \leftarrow prev \circ c$              ▷ We're decoding $prev \circ c$.
          $i \leftarrow i + 1$
      else
          Output $\perp$ and exit                     ▷ decoding failed: invalid index
      Output $T[k]$                                   ▷ Finally: output $T[k]$ and update $prev$
      $prev \leftarrow T[k]$
```

Here we present the full algorithm for LZW decompression in Algorithm 4. Note that the algorithm must handle the special case where the input contains the string $c \circ w \circ c \circ w \circ c$, where $c \circ w$ is parsed first. In the compression algorithm, $c \circ w \circ c$ is added next to the dictionary. But, the decompressor cannot add this entry to the dictionary until it knows the first character of the next string of the input. The special case handles this situation.

C Squeeze Decryption

Algorithm 5 The squeeze decryption algorithm.

```
Input: key $k_{\text{prp}}$, input stream of indices $I$
Output: output character stream $O$
  Read $iv$ from $I$
                                        ▷ Initialize PRG $G$, table $T$, and variables $prev$ and $c$
  InitializeSqueeze($k_{\text{prp}}$, $iv$, $G$, $T$, $prev$, $c$, $m$)
  while there is more input in $I$ do                 ▷ Start decompressing and decrypting stream $I$
      Read next index $i$
      Compute the pseudo-random index $r$ of the new entry   ▷ i.e., where the new $prev \circ c$ will be
      if $T[i]$ is undefined and ($i \neq r$ or $prev = \Lambda$) then
          Output $\perp$ and fail                              ▷ decoding failed: invalid index
                                                  ▷ Skip these two tests on the first iteration
      if $T[i]$ is defined and $i \neq r$ and $prev \neq \Lambda$ then    ▷ $i$ is valid and not part of a collision
          $c \leftarrow$ head($T[i]$)
          RandomInsert($prev \circ c$, $T$)
      else if $i = r$ and $prev \neq \Lambda$ then          ▷ Special case: original input was $prev \circ prev \circ c$
          $c \leftarrow$ head($prev$)
          RandomInsert($prev \circ c$, $T$)      ▷ We're decoding $prev \circ c$ and it will be in position $r = i$
      Output $T[i]$                                   ▷ Finally: output $T[i]$ and update $prev$
      $prev \leftarrow T[i]$
      RandomSwap($i$, $T$)                                   ▷ Move $prev$ to a random position
```