Compression and Information Leakage of Plaintext

John Kelsey, Certicom
(kelsey.j@ix.netcom.com)

1 Introduction

Cryptosystems like AES and triple-DES are designed to encrypt a sequence of input bytes (the plaintext) into a sequence of output bytes (the ciphertext) in such a way that the output carries no information about that plaintext except its length. In recent years, concerns have been raised about "side-channel" attacks on various cryptosystems–attacks that make use of some kind of leaked information about the cryptographic operations (e.g., power consumption or timing) to defeat them. In this paper, we describe a somewhat different kind of side-channel provided by data compression algorithms, yielding information about their inputs by the size of their outputs. The existence of some information about a compressor's input in the size of its output is obvious; here, we discuss ways to use this apparently very small leak of information in surprisingly powerful ways.

The compression side-channel differs from side-channels described in [Koc96] [KSHW00] [KJY00] in two important ways:

1. It reveals information about plaintext, rather than key material.
2. It is a property of the *algorithm*, not the implementation. That is, any implementation of the compression algorithm will be equally vulnerable.

1.1 Summary of Results

Our results are as follows:

1. Commonly-used lossless compression algorithms leak information about the data being compressed, in the size of the compressor output. While this would seem like a very small information leak, it can be exploited in surprisingly powerful ways, by exploiting the ability of many compression algorithms to adapt to the statistics of their previously-processed input data.
2. We consider the "stateless compression side-channel," based on the compression ratio of an unknown string without reference to the rest of the message's contents. We also consider the much more powerful "stateful compression side-channel," based on the compression ratio of an unknown string, given information about the rest of the message.
3. We describe a number of simple attacks based mainly on the stateless side-channel.
4. We describe attacks to determine whether some string S appears often in a set of messages, using the stateful side-channel.

5. We describe attacks to extract a secret string S that is repeated in many compressed messages, under partial chosen plaintext assumptions, using the stateful side-channel.
6. We consider countermeasures that can make both the stateless and the stateful side-channels substantially harder to exploit, and which may thus block some of these attacks.
7. We discuss the implications of these results, in light of the widespread use of compression with encryption, and the "folk wisdom" suggesting that adding compression to an encryption application will increase security.

1.2 Practical Impact of Results

Compression algorithms are widely used in real-world applications, and have a large impact on those applications' performance in terms of speed, bandwidth requirements, and storage requirements. For example, PGP and GPG compress using the Zip Deflate algorithm before encrypting, IPSec can use IPComp to compress packets before encrypting them, and both the SSH and TLS protocols support an option for on-the-fly compression.

Potential security implications of using compression algorithms are of practical importance to people designing systems that might use both compression and encryption.

The side-channel attacks described in this paper can have a practical impact on security in many situations. However, it is important to note that these attacks have little security impact on, say, a bulk encryption application which compresses data before encrypting. To a first-order approximation, the attacks in this paper are described in decreasing order of practicality. The string-extraction attacks are not likely to be practical against many systems, since they require such a specialized kind of partial chosen-plaintext access. The string-detection attacks have less stringent requirements, and so are likely to be useful against more systems. The passive information leakage attacks are likely practical to use against any system that uses compression and encryption together, and for which some information about input size is available.

In a broader sense, the results in this paper point to the need to consider the impact of any pre- or post-processing done along with encryption and authenticaton. For example, we have not considered timing channels from compression algorithms in this paper, but such channels will clearly exist for some compression algorithms, and must also exist for many other kinds of processing done on plaintext before it is sent, or ciphertext after it is received and decrypted. Similarly, anything done to the decrypted ciphertext of a message before authenticating the result is subject to reaction attacks: attacks in which changes in the ciphertext can cause different error messages or other behavior on the part of the receiver, depending on some secret information that the attacker seeks to reveal. (For decompressors which must terminate decompression with an error for some possible inputs, for example, there are serious dangers with respect to reaction attacks, or even with buffer-overrun or other related attacks.)

1.3 Previous Work

Although existence of the stateless compression side channel is obvious, we have seen very little reference to it in the literature. Nearly all published works discussing compression and encryption describe how compression *improves* the security of encryption.

One of the attendants of FSE2002 brought [BCL02] to our attention; in this article, researchers had noticed that they could use the compression ratio of a file to determine the language in which it was written in. This is the same phenomenon on which is based one of our stateless side channels.

1.4 Guide to the Paper

The remainder of this paper is arranged as follows: First, we discuss commonly-used compression methods, and how they interact with encryption. Next, we describe the side-channel which we will use in our attacks. We then consider several kinds of attack, making use of this side channel. We conclude with a discussion of various complications to the attacks, and possible defenses against them.

2 Lossless Compression Methods and the Compression Side-Channels

The goal of any compression algorithm (note: in this paper, we consider only *lossless* compression algorithms) is to reduce the redundancy of some block of data, so that an input that required R bits to encode can be written as an output with fewer than R bits. All lossless compression algorithms work by taking advantage of the fact that not all messages of R bits are equally likely to be sent. These compression algorithms make a trade-off: they effectively encode the higher probability messages with fewer bits, while encoding the lower probability messages with more bits. The compression algorithms in widespread use today typically use two assumptions to remove redundancy: They assume that characters and strings that have appeared recently in the input are likely to recur, and that some values (strings, lengths, and characters) are more likely to occur than others. Using these two assumptions, these algorithms are effective at compressing a wide variety of commonly-used data formats.

Many compression algorithms (and specifically, the main one we will consider here) make use of a “sliding window” of recently-seen text. Strings that appear in the window are encoded by reference to their position in the window. Other compression algorithms keep recently-seen strings in an easily-searched data structure; strings that appear in that structure are encoded in an efficient way.

Essentially all compression algorithms make use of ways to efficiently encode symbols (characters, strings, lengths, dictionary entries) of unequal frequency, so that commonly-occurring symbols are encoded using fewer bits than rarely-occurring symbols.

For the purposes of this paper, it is necessary to understand three things about these compression functions:

1. At any given point in the process of compressing a message, there are generally many different input strings of the same length which will compress to different lengths. This inherently leaks information about these input strings.
2. The most generally useful compression algorithms encode the next few bytes of input in different ways (and to different lengths), depending on recently-seen inputs.
3. While a single "pass" of a compression algorithm over a string can leak only a small amount of data about that string, multiple "passes" with different data appearing before that string can leak a great deal of data about that string.

This summary necessarily omits a lot of detail about how compression algorithms work. For a more complete introduction to the techniques used in compression algorithms, see [Sal97a] or [CCF01a].

2.1 Interactions with Encryption

Essentially all real-world ciphers output data with no detectable redundancy. This means that ciphertext won't compress, and so if a system is to benefit from compression, it must compress the information before it is encrypted.

The "folk wisdom" in the cryptographic community is that adding compression to a system that does encryption adds to the security of the system, e.g., makes it less likely that an attacker might learn anything about the data being encrypted. This belief is generally based on concerns about unicity distance, keysearch difficulty, or ability of known- or chosen-plaintext attacks. We believe that this folk wisdom, though often repeated in a variety of sources, is not generally true; adding compression to a competently designed encryption system has little real impact on its security. We base this on three observations:

Unicity distance is irrelevant. The unicity distance of an encryption system is the number of bits of ciphertext an attacker must see before he has enough information that it is even theoretically possible to determine the key. Compression algorithms, decreasing the redundancy of plaintexts, clearly increase unicity distance. However, this is irrelevant for practical encryption systems, where a single 128-bit key can be expected to encrypt millions of bytes of plaintext.

Keysearch difficulty is only slightly increased. Since most export restrictions on key lengths have gone away, we can expect this to become less and less relevant over time, as existing fielded algorithms with 40- and 56-bit key lengths are replaced with triple-DES or AES. At any rate, for systems with keys short enough for brute force searching, adding general-purpose compression algorithms to the system seems like a singularly unhelpful way to fix the problem. Standard compression algorithms usually include fixed headers, and tend to be pretty predictable in their first few bytes of output.

It seems unlikely that adding such a compression algorithm, even with fixed headers removed, increases the difficulty of keysearch by more than a factor of 10 to 100. Switching to a stronger cipher is a far cheaper solution that actually solves the problem.

Standard algorithms not that helpful. Compression with some additional features to support security (such as a randomized initial state) can make known-plaintext attacks against block ciphers much harder. However, off-the-shelf compression algorithms provide little help against known-plaintext attacks (since an attacker who knows the compression algorithm and the plaintext knows the compressor output). And while chosen-plaintext attacks can be made much harder by specially designed compression algorithms, they are also made much harder, at far lower cost, by the use of standard chaining modes.

In summary, compression algorithms add very little security to well-designed encryption systems. Such systems use keys long enough to resist keysearch attack and chaining modes that resist chosen-plaintext attack. The real reason for using compression algorithms isn't to increase security, but rather to save on bandwidth and storage. As we will disucuss below, this real advantage needs to be balanced against a (mostly academic) risk of attacks on the system, such as those described below, based on information leakage from the compression algorithm.

3 The Compression Side-Channel and our Attack Model

In this section, we describe the compression side channel in some detail. We also consider some situations in which this side channel might leak important data.

Any lossless compression algorithm must compress different messages by different amounts, and indeed must expand some possible messages. The compression side channel we consider in this paper is simply the different amount by which different messages are compressed. When an unknown string S is compressed, and an attacker sees the input and output sizes, he has almost certainly learned only a very small amount about S. For almost any S, there will be a large set of alternative messages of the same length, which would also have had the same size of compressor output. Even so, *some* small amount of information is leaked by even this minimal side-channel. For example, an attacker informed that a file of 1MB had compressed to 1KB has learned that the original file must have been *extremely* redundant.

Fortunately (for cryptanalysts, at least), compression algorithms such as LZW and Zip Deflate adapt to the data they are fed. (The same is true of many other compression algorithms, such as adaptive markov coding and Burrows-Wheeler coding, and even adaptive Huffman coding of symbols.) As a message is processed, the state of the compressor is altered in a predictable way, so that strings of symbols that have appeared earlier in the message will be encoded more efficiently than strings of symbols that have not yet appeared in the message. This allows an enormously more powerful side-channel when the unknown string

S is compressed with many known or chosen prefix strings, $P_0, P_1, ..., P_{n-1}$. Each prefix can put the compressor into a different state, allowing new information to be extracted from the compressor output size in each case. Similarly, if a known or chosen set of suffixes, $Q_0, Q_1, ..., Q_{n-1}$ is appended to the unknown string S before compression, the compressor output sizes that result will each carry a slightly different piece of information about S, because those suffixes with many strings in common with S will compress better than other suffixes, with fewer strings in common with S. This can allow an attacker to reconstruct all of S with reasonably high probability, even when the compressor output sizes for different prefixes or suffixes differ only by a few bytes. In this situation, it is quite possible for an attacker to rule out all incorrect values of S given enough input and output sizes for various prefixes, along with knowledge or control over the prefix values. Further, an attacker can build information about S gradually, refining a partial guess when the results of each successive compressor output are seen.

A related idea can be used against a system that compresses and encrypts, but does not strongly authenticate its messages. The effect of altering a few bytes of plaintext (through a ciphertext alteration) will be very much dependent on the state of the decompressor both before and after the altered plaintext bytes are processed. The kind of control exerted over the compressor state is different, but the impact is similar. However, we do not consider this class of attack in this paper.

3.1 Assumptions and Models

We will make the following assumptions in the remainder of this paper:

1. Each message is processed by first compressing it, then encrypting it.
2. The attacker can learn the precise compressor output length from the ciphertext.
3. The attacker somehow knows the precise input length, or (in some cases) at least the approximate input length.

In the sections that follow, we will consider three basic classes of attacks: First, we will consider purely passive attacks, where the attacker simply observes the ciphertext length and compression ratio, and learns information that should have been concealed by the encryption mechanism. Second, we will consider a kind of limited chosen-plaintext attack, in which the attacker attempts to determine whether and approximately how often some string appears in a set of messages. Third, we will consider a much more demanding kind of chosen-plaintext attack, in which the attacker must make large numbers of chosen or adaptive-chosen plaintext queries, in hopes of extracting a whole secret string.

4 Data Information Leakage

In this section, we consider purely passive attacks; ways that an attacker can learn some information he should not be able to learn, by merely observing the ciphertexts and corresponding compression ratio. One general property of these attacks is that they are quite hard to avoid, without simply eliminating compression from the system. However, it is also worth noting that most of these attacks are not particularly devastating under most circumstances.

4.1 Highly Redundant Data

Consider a large file full of binary zeros or some other very repetitive contents. Encrypting this under a block cipher in ECB-mode would reveal a lot of redundancy; this is one reason why well-designed encryption systems use block ciphers in one of the chaining modes. Using CBC- or CFB-mode, the encrypted file would reveal nothing about the redundancy of the plaintext file.

Compressing before encryption changes this behavior. Now, a highly-redundant file will compress extremely well. The very small ciphertext will be sufficient, given knowledge of the original input size, to inform an attacker that the plaintext was highly redundant.

We note that this information leak is not likely to be very important for most systems. However:

1. Chaining modes prevent this kind of information leakage, and this is, in fact, one very good reason to use chaining modes with block ciphers.
2. In some situations, leaking the fact that highly-redundant data is being transmitted may leak some very important information. (An example might be a compressed, encrypted video feed from a surveilance camera–an attacker could watch the bandwidth consumed by the feed, and determine whether the motion of his assistant trying to get past the camera had been detected.)

4.2 Leaking File or Data Types

Different data formats compress at different ratios. A large file containing ASCII-encoded English text will compress at a very different ratio from a large file containing a Windows executable file. Given knowledge only of the compression ratio, an attacker can thus infer something about the kind of data being transmitted. This is not so trivial, and may be relevant in some special circumstances.

This may be resisted by encoding the data to be transmitted in some other format, at the cost of losing some of the advantage of compression.

4.3 Compression Ratio as a Checksum

Consider a situation where an attacker knows that one of two different known messages of equal length is to be sent. (For example, the two message might be something like "DEWEY DEFEATS TRUMAN!" or "TRUMAN DEFEATS DEWEY!".) If these two messages have different compression ratios, the attacker can determine precisely which message was sent. (For this example, Python's ZLIB compresses "TRUMAN DEFEATS DEWEY!" slightly better than "DEWEY DEFEATS TRUMAN!")

More generally, if the attacker can enumerate the set of possible input messages, and he knows the compression algorithm, he can use the length of the input, plus the compression ratio, as a kind of checksum. This is a very straightforward instance of the side-channel; an attacker is able, by observing compression ratios, to rule out a subset of possible plaintexts.

4.4 Looped Input Streams

Sometimes, an input stream may be "looped," so that after R bytes, the message begins repeating. This is the sort of pattern that encryption should mask, and without compression, using a standard chaining mode will mask it. However, if the compression ratio is visible to an attacker, he will often be able to determine whether or not the message is looping, and may sometimes be able to determine its approximate period.

There are two ways the information can leak. First, if the period of the looping is shorter than the "sliding window" of an LZ77-type compression algorithm, the compression ratio will suddenly become very good. Second, if the period is longer than the sliding window, the compression ratios will start precisely repeating. (Using an LZW-type scheme will leave the compression ratios improving each time through the repeated data, until the dictionary fills up.)

5 String Presence Detection Attacks

The most widely used lossless compression algorithms adapt to the patterns in their input, so that when those patterns are repeated, those repetitions can be encoded very efficiently. This allows a whole class of attacks to learn whether some string S is present within a sequence of compressed and encrypted messages, based on using either known input data (some instances where S is known to have appeared in messages) or chosen input (where S may be appended to some messages before they're compressed and encrypted).

All the attacks in this section require knowledge or control of some part of a set of messages, and generally also some knowledge of the kind of data being sent. They also all require knowledge of either inputs or compressor outputs, or in some cases, compression ratios.

5.1 Detecting a Document or Long String with Partial Chosen Plaintext

The attacker wants to determine whether some long string S appears often in a set of messages $M_0, M_1, ..., M_{N-1}$.

The simplest attack is as follows:

1. The attacker gets the compressed, encrypted versions of all of the M_i. From this he learns their compressed output lengths.

2. The attacker requests the compressed, encrypted versions of $M_i' = M_i, S$, for all M_i. That is, he requests the compressed and encrypted results of appending S to each message.
3. The attacker determines the length of S after compression with the scheme in use.
4. The attacker observes $M_i' - M_i$. If these values average substantially less than the expected length of S after compression, it is very likely that S is present in many of these messages.

5.2 Partial Known Input Attack

A much more demanding and complicated attack may be possible, given only the leakage of some information from each of a set of messages. The attacker can look for correlations between the appearance of substrings of S in the known part of each message, and the compressed length of the message; based on this, he can attempt to determine whether S appears often in those messages. This attack is complicated by the fact that the appearance of substrings of S in the known part of the message may be correlated with the presence of S in the message. (Whether it is correlated or not requires more knowledge about how the messages are being generated, and the specific substrings involved. For example, if S is "global thermonuclear war", the appearance of the substring "thermonuclear" is almost certainly correlated with the appearance of S later in the message.)

A more useful version of an attack like this might be a case where several files are being combined into an archive and compressed, and the attacker knows one of the files. Assuming the other files aren't chosen in some way that correlates their contents with the contents of the known file, the attacker can safely run the attack.

6 String Extraction Attacks

In this section, we consider ways an attacker might use the compression side channel to extract some secret string from the compressor inputs. This kind of attack requires rather special conditions, and so is much less practical than the other attacks considered above. However, in some special situations, these attacks could be made to work. More importantly, these attacks demonstrate a separate path for attacking systems, despite the use of very strong encryption.

The general setting of these attacks is as follows: The system being attacked has some secret string, S, which is of interest to the attacker. The attacker is permitted to build a number of requested plaintexts, each using S, without ever knowing S. For example, the attacker may choose a set of N prefixes, $P_{0,1,\ldots,N-1}$, and request N messages, where the ith message is $P_i||S$.

6.1 An Adaptive Chosen Input Attack

Our first attack is an adaptive chosen input attack. We make a guess about the contents of the first few characters of the secret string, and make a set of queries based on this guess. The output lengths of the results of these queries should be smaller for correct guesses than for incorrect guesses.

We construct our queries in the form

$$\text{Query} = \text{prefix} + \text{guess} + \text{filler} + \text{prefix} + S$$

where

Query is the string which the target of the attack is convinced to compress and encrypt.
prefix is a string that is known not to occur in S.
filler is another string known not to occur in S, and with little in common with $prefix$.
S is the string to be recovered by the attacker.

The idea behind this attack is simple: Suppose the prefix is 8 characters long, and the guess is another 4 characters long. A correct guess guarantees that the query string contains a repeated 12-character substring; a good compression algorithm (and particularly, a compression scheme based on a sliding window, like Zip Deflate) will encode this more efficiently than queries with incorrect guesses, which will contain a string with slightly less redundancy. When we have a good guess, this attack can be iterated to guess another four digits, and so on, until all of S has been guessed.

Experimental Results We implemented this attack using the Python Zlib package, which provides access to the widely-used Zip Deflate compression algorithm. The search string was a 16-digit PIN, and the guesses were four (and later five) digits each. Our results were mixed: it was possible to find the correct PIN using this attack, but we often would have to manually make the decision to backtrack after a guess. There were several interesting complications that arose in implementing the attack:

1. The compression algorithm is a variant of a sliding-window scheme, in which it is not always guaranteed that the longest match in the window will be used to encode a string. More importantly, this is a two-pass algorithm; the encoding of strings within the sliding window is affected by later strings as well as earlier ones, and this can change the output length enough to change which next four digits appear to be the best match to S[Whi02].
2. Some guesses themselves compress very well. For example, the guess "0000" compresses quite well the first time it occurs.
3. The actual "signal" between two close guesses (e.g., "1234" and "1235") is very close, and is often swamped by the "noise" described above.
4. To make the attack work reasonably well, it is necessary to make each piece of the string guessed pretty long. For our implementation, five digits worked reasonably well.
5. Some backtracking is usually necessary, and the attack doesn't always yield a correct solution.

6. It turns out to also be helpful to add some padding at the end of the string, to keep the processing of the digits uniform.

All of these problems appear to be pretty easy to solve given more queries and more work. However, we dealt with them more directly by developing a different attack—one that requires only chosen-plaintext access, not adaptive chosen plaintext access.

6.2 A Chosen Input Attack

The adaptive chosen input attack seems so restrictive that it is hard to see how it might be extended to a simple chosen or known plaintext attack. However, we can use a related, but different approach, which gives us a straightforward chosen input attack.

The attack works in two phases:

1. Generate a list of all possible subsequences of the string S, and use the compression side-channel to put them in approximate order of likelihood of appearing in S.
2. Piece together the subsequences that fit end-to-end, and use this to reconstruct a set of likely candidate values for S.

The subsequences can be tested in the simplest sense by making queries of the form

$$\text{Query} = \text{Guess} + S$$

However, to avoid interaction between the guess and the first characters of S, it is useful to include some filler between them.

Experimental Results We were able to implement this attack, with about a 70% success rate for pseudorandomly-generated strings S of 16 digits each, using the Python Zlib. The attack generates a list of the 20 top candidates for S, and we counted it as a success if any of those 20 candidates was S.

There were several tricks we discovered in implementing this attack:

1. In building the queries, it was helpful to generate padding strings before the guessed subsequence, between the guess and the string S, and after the string.
2. It was very helpful to generate several different padding strings, of different lengths, and to sum the lengths of the compressed strings resulting from several queries of the same guess. This tended to average out some of the "noise" of the compression algorithm.
3. There are pathological strings that cause the attack to fail. For example, the string "0000000123000000" will tend to end up with guesses that piece together instances of "00000".

7 Caveats and Countermeasures

The attacks described above make a number of simplifying assumptions. In this section, we will discuss some of those assumptions, and the implications for our attacks when the assumptions turn out to be false. We will also consider some possible countermeasures.

7.1 Obscuring the Compressor Input Size

The precise size of the input may be obscured in some cases. Naturally, some kind of information about relative compression ratios is necessary for the attack to work. However, approximate input information will often be good enough, as when the compression ratio is being used as the side channel. An approximate input size will lead to an approximate compression ratio, but for any reasonably large input, the difference between the approximate and exact compression ratios will be too small to make any difference for the attack.

One natural way for an attacker to learn approximate input size is for the process generating the input to the compressor to have either some known constant rate of generating input, or to have its operations be visible (e.g., because it must wait for the next input, which may be observed, before generating the next output).

7.2 Obscuring the Compressor Output Size

Some encryption modes may automatically pad the compresor output to the next full block for the block cipher being used. Others may append random padding to resist this or other attacks. For example, some standard ways of doing CBC-mode encryption include padding to the next full cipher block, and making the last byte of the padding represent the total number of bytes of padding used. This gives the attacker a function of the compressor output size, $\lceil(\mathrm{len}+1)/\mathrm{blocksize}\rceil \times \mathrm{blocksize}$. These may slightly increase the amount of work done during our attacks, but don't really block any of the attacks.

A more elaborate countermeasure is possible. A system designer may decide to reduce the possible leakage through the compressor to one bit per message, as follows:

1. Decide on a compression ratio that is acceptable for nearly all messages, and is also attainable for nearly all messages.
2. Send the uncompressed version of any messages that don't attain the desired compression ratio.
3. Pad out the compressor output of messages that get at least the desired compression ratio, so that the message effectively gets the desired compression ratio.

This is an effective countermeasure against some of our attacks (for example, it makes it quite hard to determine which file type that compresses reasonably well has been sent), but it does so at the cost of losing some compression performance. Against our chosen-input attacks, this adds a moderate amount of difficulty, but doesn't provide a complete defense.

7.3 Obscuring the Compressor Internal State

It is possible to obscure the internal state of the compressor, in a number of simple ways, including initializing the compressor in a random state, or inserting occasional random blocks of text during the compression operation. In either case, this can cause problems with some of our attacks, because of the lack of precise information about the state of the compressor when an unknown string is being processed. General compression ratios are unlikely to be affected strongly by such countermeasures, however, so the general side channel remains open.

7.4 Preprocessing the Text

The text may be preprocessed in such a way that compression is affected in a somewhat unpredictable way. For example, it is easy to design a very weak stream cipher, which generates a keystream with extremely low Hamming weight. Applying this kind of stream cipher to the input before compression would degrade the compression slightly, in a way not known ahead of time by any attacker. By allowing the Hamming weight of the keystream to be tunable, we could get tunable degradation to the compression.

8 Conclusions

In this paper, we have described a side-channel in widely-used lossless compression algorithms, which permit an attacker to learn some information about the compressor input, based only on the size of the compressor output and whatever additional information about other parts of the input may be available.

We have discussed only a small subset of the available compression algorithms, and only one possible side channel (compression ratio). Some interesting directions for future research include:

1. Timing side-channels for compression algorithms.
2. Attacking other lossless compression algorithms, such as adaptive Huffman encoding, adaptive Markov coders, and Burrows-Wheeler block sorting (with move-to-front and Huffman or Shannon-Fano coding) with this side channel. Adaptive Huffman and Markov coders can be attacked using techniques very similar to the ones described above. Burrows-Wheeler block sorting appears to require rather different techniques, though the same side channels clearly exist and can be exploited.
3. Attacking lossy compression algorithms for image, sound, and other data with this side channel.

4. Attacking lossy image compression by trying to use disclosed parts of a compressed image to learn undisclosed parts of the same image, as might be useful for redacted scanned documents.
5. Reaction attacks against decompressors, such as might be useful when a system cryptographically authenticates plaintext, then compresses and encrypts it. This might lead either to software faults (a change in ciphertext leading to a buffer overrun, for example) or to more general leakage of information about the encryption algorithm or plaintext.

9 Acknowledgements

The author wishes to thank Paul Crowley, Niels Ferguson, Andrew Fernandes, Pieter Kasselman, Yoshi Kohno, Ulrich Kuehn, Susan Langford, Rene Struik, Ashok Vadekar, David Wagner, Doug Whiting, and the many other people who made helpful comments after seeing these results presented at Certicom, at the Crypto 2001 Rump Session, and at FSE2002. The author also wishes to thank the anonymous referees for several useful suggestions that improved the paper.

References

[BCL02] Benedetto, Caglioti, and Loreto, *Physical Review Letters*, 28 January 2002.

[CCF01a] Usenet group comp.compression FAQ file, available at `http://www.faqs.org/faqs/compression-faq/`, 2001.

[KJY00] Kocher, Jaffe, Jun, “Differential power analysis: Leaking secrets,” in *Advances in Cryptology – CRYPTO’99*, Springer-Verlag, 1999

[Koc96] Kocher, “Timing Attack on Implementations of Diffie-Hellman, RSA, DSS and other systems,” in *Advances in Cryptology - CRYPTO ’96*, Springer-Verlag, 1996.

[KSHW00] Kelsey, Schneier, Wagner, Hall, “Side Channel Cryptanalysis of Product Ciphers,” in *Advances in Cryptology–ESORICS 96*, Springer-Verlag, 1996.

[Sal97a] David Salomon, *Data Compression: The Complete Reference*, Springer-Verlag, 1997.

[Whi02] Doug Whiting, personal communication, 2002.