A Toolkit for Modeling and Compressing Audit Data

Chapman Flack, Mikhail J. Atallah
COAST Laboratory
Purdue University
West Lafayette, IN 47907

COAST-TR-98-20

March 10, 1999

Abstract

System administrators face trade-offs concerning the volume of audit data to collect and retain. Not all approaches have easily quantified costs, but lossless compression offers an adjustable trade-off of storage for compute time. Compression techniques designed into the data format can complicate software that consumes the data, and are not adjustable to suit the needs of diverse sites. General-purpose compression tools permit some adjustment, but cannot exploit sophisticated models of the data. The toolkit described here simplifies tailoring compression tools to the properties of the data at any time after the data format is specified. Using the toolkit, a few days of work defining models can achieve compression 13% better than `gzip` on an existing commercial audit format, with many known properties of the data remaining to be exploited by refinements of the models for still better compression. A customized compression tool could also be designed to permit recovery of data from a compressed stream without decompressing the entire stream.

1 Introduction

Computer systems store and process information whose disclosure, alteration, or unavailability could have economic or strategic costs. Audit logs may help detect orchestrated or accidental events that could compromise confidentiality, integrity, or availability. Log records may suggest what went wrong and how, the extent of the damage, how to recover, and how to prevent a recurrence. The log may help determine accountability for negligent or deliberate acts, or provide valuable evidence if legal action is appropriate. It may be used retrospectively to reconstruct and understand incidents that were not anticipated when logging was established, and it may be combined with other sources of information in the investigation of a larger event in which the computer system played only an incidental role. Administrative choices about the resources to be dedicated to auditing are complicated by the variety of uses to which audit logs may be put.

Audit logs contain records of simple events frequently seen in normal operation. Records revealing questionable activity, if any, appear amid long sequences of unremarkable events. The rate of log growth can make significant demands on computing and storage: Mounji[10] reports that systems with modest-sized user communities can collect hundreds of megabytes of audit data per day, and the peak rate observed for a single machine in this laboratory was equivalent to 97 megabytes per day during a period of heavy use.

Administrators have some options in managing the storage demands. *Preselection criteria* allow the logging of only selected subsets of auditable events, and the retention period for old logs can be set by policy. Both options have costs. The cost of preselection is a risk that insufficient information may be available to understand an event after the fact. The cost of a short retention period is the risk that no information may be available if an unusual event is questioned some time after it occurred.

In setting selection and retention policy, the administrator must negotiate a trade-off of storage capacity against these risks, which are difficult to quantify in economic terms. Another option, lossless compression of audit logs, also involves a trade-off, but the trade is storage capacity against compute time, whose value is easily quantified. By pairing a generic compression technique with more or less sophisticated models of the data, compute cost and compression ratio can be balanced. The nine compression levels of `gzip`[1, 2] illustrate the idea, but this work concerns the construction of models to achieve better compression than `gzip`. The techniques and tools described can be applied to other sources of data that, like audit logs, are highly structured. As described in Section 7, they may also be used to compress data such that portions of the compressed stream can be recovered without decompressing from the beginning.

2 Techniques in current use

Auditing systems now available include design features intended to keep storage demands reasonable. Section 2.1 describes two examples from current practice of addressing space concerns directly in the canonical concrete syntax of the audit log. *Abstract syntax* is used here to denote the inherent structure and content of a data stream, without regard to any encoding of that stream on a physical medium. A *concrete syntax* specifies the details of representation on a physical medium, defining such things as delimiters or length encodings by which data described by the abstract syntax can be recoverably stored or transmitted in a concrete form. Many concrete syntaxes can be defined for a given abstract syntax, so *canonical* distinguishes the concrete syntax that is documented as the audit log format, and that developers of log-consuming tools are expected to implement. Section 2.2 describes another approach, using a simpler canonical concrete syntax and a general-purpose compression tool, whose output may be regarded as an alternative concrete representation of the same abstract syntax.

2.1 Hand-optimized concrete syntax

The Solaris Basic Security Module, a commercial system in current use[9], and the Common Intrusion Specification Language (CISL)[3], a cross-platform effort now in development, both illustrate the approach of addressing space concerns in the canonical concrete syntax design.

2.1.1 Solaris BSM

The BSM audit log is a sequence of audit 'tokens' constructed from the internal binary representations of kernel data. Some of these representations encode more than one value (for example a device driver and a unit number) into a single integral field; others are bit-mapped collections of boolean flags. In some cases the flags are overloaded so the correct interpretation depends on values of other flags or fields.

2.1.2 Common Intrusion Specification Language (CISL)

The Common Intrusion Detection Framework (CIDF)[13] working group was convened by the Defense Advanced Research Projects Agency with representation from industry, academia, and research organizations. CISL, a component of the CIDF effort, is defined at a high level, but Section 5 of the document is devoted to a binary concrete syntax. An example of the compression techniques included in the CISL design is the encoding of length shown in Figure 1. The encoding of any variable-size object is preceded by a length,

$$
\begin{array}{lcl}
l < 2^{8} & \Rightarrow & \boxed{1}\,\boxed{\;l\;} \\
l < 2^{16} & \Rightarrow & \boxed{2}\,\boxed{\quad l \quad} \\
l < 2^{32} & \Rightarrow & \boxed{4}\,\boxed{\qquad\quad l \qquad\quad}
\end{array}
$$

Figure 1: Length encoding in the Common Intrusion Specification Language

which can be up to 32 bits. To make the encoding more compact, only one octet is transmitted if the length is less than 256, or only two if it is less than 16 384. The length itself is then preceded by an octet value 1, 2, or 4, indicating the number of octets occupied by the length value.

2.2 General-purpose compression tools

Although the hand-optimization techniques can reduce the space consumed by an audit log, they must be applied during the initial design of the format. At that stage of design, the space considerations compete for attention with other design goals, which should include adequacy of the captured data for the purposes of audit, and simplicity of implementation for programs that will need to consume the audit log. By contrast, a general-purpose compression tool can be placed underneath any file format even if it was not anticipated in design. It can compress a format that was designed with no attention to space efficiency, or one whose original design does not yield the space efficiency needed for a new application. If compression is available as a separate layer and decoupled from initial format design, designers can attend to the goals of usability of the log and simplicity of tool construction.

Intuitively, the structure and repetition in audit data contribute to compressibility. Intuition is bolstered by the performance of completely general-purpose data compression tools such as `gzip`, which can reduce Solaris BSM audit data in this laboratory to 6–7 percent of its original size without any special knowledge of what it is compressing. This may be adequate for many applications; where better compression is needed, a tool that is tailored to the characteristics of the data can be used.

3 A systematic approach

An audit format fed to a generic compression tool creates a clear layer boundary but limits the compression ratio. A hand-optimized format, while it can exploit known characteristics of the data, destroys the layer distinction, can complicate tools that consume the data, and usually achieves compression far from optimal. Consider the CISL length encoding. It is reasonable to suppose that the most common lengths encoded fall in the 0 to 255 range, expressible in one octet. CISL uses a second octet for the encoding length, expanding the most frequent lengths by 100 percent.

A systematic approach to the problem begins with a general purpose algorithm, based on sound compression theory, that can be tailored to the characteristics of the input data without requiring the data format be tailored to it.

3.1 Information rate and capacity

C. E. Shannon[12] considered the problem of determining the information-carrying capacity of an arbitrary communication channel and the information content of a message stream. His measure of channel capacity depends on the alphabet of symbols that can be transmitted over the channel, the relative probability of each symbol, and the time (more generally, cost) to transmit each symbol. Entropy, his measure of information content in the symbols of a message, is also a function of the number and relative frequencies of the symbols. For a channel with alphabet and costs given, there is an assignment of probabilities to the symbols that will minimize the average cost of transmitting a message. For the case where all symbols have the same cost, the minimum is achieved when all are equiprobable. If messages are to be sent whose symbol probabilities have any other distribution, transmission cost can be reduced by first recoding the messages into a sequence of symbols with probability distributions near the optimal. The benefit to be expected from recoding depends on the relative entropy of the original message stream, defined in the uniform-cost case as the ratio of the original message stream entropy to the entropy of a language over the same alphabet but with the symbols equiprobable.

The recoding process, which Shannon called a discrete transducer, is specified by its input and output alphabets and probability distributions. If the output characteristics of one transducer match the input characteristics of another, the two can be composed and the result is also a transducer. If the composition of two transducers reproduces the original message stream, one is the inverse of the other and both are said to be nonsingular. Shannon outlined the design of a nonsingular discrete transducer for any message stream, given a model for the statistical properties of the input, and the statistical properties desired for the output.

The compression problem, as usually formulated, is none other than the optimal utilization of a storage channel whose two symbols, 0 and 1, have equal cost. It also can be solved by a discrete transducer if a good model of the statistical properties of the input can be constructed. Here is the foundation of the earlier intuitive argument that the known regularities of audit data can be exploited to achieve good compression. At most points in an audit log the possible next symbols are far from equiprobable, so the relative entropy of the log message stream is low and the expected benefit of recoding is high. The probability models needed for recoding can exploit many known characteristics of the audited system. Unlike a general text compression problem, it is not necessary to inductively determine the best structure and order of a Markov model given only the data. Rather, models can be crafted to reflect the known logic of the system routines that produce the audit data.

3.2 Exploitable redundancy in audit data

3.2.1 Grammatical structure

Because the audit log is generated by an automaton, it can be described by a grammar. The grammar is at least partially documented, and a parser can be built. The grammar strictly constrains the possible choice of next symbol at any point in the log, and that information is available to a parser. The relative probabilities of symbols that cannot appear are known to be zero, and the parser can record the frequencies of the symbols that have appeared for each production.

3.2.2 Persistent attributes of subjects and objects

Audit logs contain frequent references to subjects, with user identities, privileges, and other properties, and objects such as files, with sizes, owners, permissions, and types. Usually, attributes of a given subject or object will have the same values they had when that subject or object was last seen; that is, those values will have much higher probability than others.

3.2.3 Constraints on operations, operands, and results

Most records in an audit log represent system calls, their parameters, and the results returned. Knowledge of the system logic can establish that certain combinations of operator and operand are invalid and so unlikely to appear; if they do appear (say, a read on a file descriptor that has not been opened), the probability of a specific error return becomes 1.0.

3.2.4 Distributions of field values

Many fields in audit records have peculiarly distributed values, and the distributions can be exploited. For example, the timestamps of audit records themselves are monotonically increasing, with a very small frequency of exceptions resulting from clock changes. Some combinations of file permissions are quite common while others are unheard of. At a given site, certain IP addresses will be especially common. Sizes of files have interesting distributions that have been studied[7, 11]. Many such examples become apparent through familiarity with the system API and internals.

3.3 How sophisticated should a model be?

It appears that a model can be constructed that will predict the probabilities of audit elements using knowledge of the audited system, its configuration, and past events. A very sophisticated model would have data structures and operations very similar to those of the kernel being modeled, and so would be comparable in size and complexity. Such a model might also refer to a database of objects from the file system to record persistent attribute values. This model could achieve excellent compression, rarely transmitting a symbol unless an anomaly is detected. It might, however, require several megabytes of storage and expensive computation and be unjustifiable for many applications. A model can begin with knowledge of the actual kernel functions being modeled, and deliberate simplifications can be introduced to achieve a desired trade of speed or space for compression performance; or very simple initial approximations can be refined to reach the same point. Finding the appropriate models for an audit source is therefore an experimental process, and would be aided by a framework for quickly implementing, testing, and comparing a variety of models.

4 A toolkit for evaluating models

This section describes a toolkit designed to speed the construction, evaluation, and refinement of models for the compression of data with known structure. The toolkit is written in the Java language[5], and uses the parser generator JavaCC[8] for modelling an input grammar. The inheritance features of Java provide a natural way to build sophisticated models quickly by extension and combination of simpler, existing models. The toolkit includes a collection of simple models from which application-specific models can be built.

4.1 Arithmetic coding engine

Shannon's arithmetic technique for constructing a nonsingular discrete transducer has been refined and is now known as arithmetic coding. A significant refinement is to bound the precision required, so the coding computations can be implemented in fast integer arithmetic. Of the published algorithms that do so, the one used in this toolkit is that described in [14]. Subsequent refinements (e.g., [6]) have improved coding speed by such techniques as approximate multiplication and division with only a slight loss in compression ratio. Those algorithms were not used in this work because they tend to place additional constraints on models. The chosen algorithm works with any model that can accept symbols and produce probabilities. This work is intended to support rapid development of experimental models, so the requirements for models were kept modest. Production use of these compression techniques will likely require the more optimized coding engines and models adapted to them.

In the most abstract description of arithmetic coding, the operation of encoding a symbol is to narrow an interval that is initially $[0,1)$ before any symbol has been encoded. Let this interval, which represents the entire sequence of symbols already encoded, be called the current coder interval.

For each symbol in the message, the set of symbols possible in that position are associated with subintervals partitioning $[0,1)$, the width of each subinterval proportional to the probability of the corresponding symbol. For example, the three symbols a, b, c, might be possible and associated with the intervals $[0,0.4)$, $[0.4,0.9)$, and $[0.9,1)$, respectively. This association of symbols with subintervals is a model. To encode the symbol b, narrow the current coder interval to the corresponding subinterval of the coder interval. If the current coder interval is $[0,0.4)$, as it might be after encoding a single a from the same model, after next encoding a b the interval will be $[0.16,0.36)$. When the entire message has been encoded, the shortest binary fraction falling within the final coder interval represents the entire sequence of symbols.

In practice, integer arithmetic is used for speed, and the initial interval is $[0,N]$ for a large integer N. The interval does not become arbitrarily small; rather, leading bits of the output "fraction" are shifted off and transmitted as soon as they become uniquely determined, and the interval is scaled up. Details may be found in [14].

In the abstract description, every set of symbols is modeled as a partition of $[0,1)$ and the encoding operation requires a linear transformation to map that standard interval onto the current coder interval before selecting the new subinterval. Because the transformation is required in any case, it is not necessary that every model be a partition of the same standard interval, and in the practical implementation no such requirement is enforced. Each model is free to define its own standard interval $[0,m]$ and partition it into subintervals. To encode a symbol, the encoder needs the scale m of the interval used by the model, as well as the endpoints $[l,u]$ of the proper subinterval. A model that simply counts frequencies can easily interface to this coder: if the model has seen 8 a's, 10 b's, and 2 c's, it needs only the cumulative frequencies to encode b as $l=8, u=18, m=20$, equivalent to the example given on the real interval $[0,1)$ above.

4.2 Encoder and Decoder in Java

The class `Encoder` encapsulates arithmetic encoding. Its constructor requires an `OutputStream` onto which the encoded bit stream will be written. The fundamental methods are:

encode(l,u,m) — Encode the symbol corresponding to subinterval $[l,u]$ of model interval $[0,m]$. This method is generally called by a model, not directly by an application. Method signatures exist for three `long` arguments and for three `int` arguments. They behave equivalently; the precision of the `Encoder` may determine which is more efficient.

doneEncoding() — Called after all symbols have been encoded, this method outputs enough additional bits to specify a number that falls in the final coder interval. A call to `doneEncoding` does *not* ensure that a decoder will be able to determine the message ends here. That must be obvious from the grammar of the transmitted message, or from an end-of-message symbol encoded before `doneEncoding` is called.

flush() — Called after `doneEncoding`, this method outputs enough extra bits to fill out a byte, then calls the `flush` method of the underlying `OutputStream`.

Some supporting methods are also available:

maxFrequency() The precision of the encoder places an upper bound on the size m of the standard interval used by any model. Every model to be used with the coder should call `maxFrequency` in its initialization to find out what that bound is.

bitsSoFar(reset) This method returns the number of bits of output attributable to the symbols encoded so far, then resets the count to zero if `reset` is `true`. The count is returned as a `double` and includes any fractional bits, not yet output, held in the coder's internal state. Surrounding the `encode` of a symbol with calls to `bitsSoFar` will return an accurate idea of that symbol's contribution to the output, even if the contribution was a fraction of a bit. `bitsSoFar` calls `java.lang.Math.log`, a potentially expensive function, but the bookkeeping done in `encode` to support it uses only integer arithmetic; with this design, most of the overhead of accounting is avoided in programs that do not use `bitsSoFar` or use it infrequently.

suppress([flag]) Returns the current *suppress* flag, then sets it to `flag`. If the optional `flag` is not present, simply returns the current value. When the *suppress* flag is true, `encode` will produce no output. It will, as always, throw an `UnrepresentableException` if the specified interval is of zero width, but will otherwise return without effect. This ability to "trial-encode" a symbol is used by piecewise models, described below.

The `Encoder` class is abstract; it is implemented by the classes `Encoder32` and `Encoder64`, which differ in the bits of precision used internally (and in the value returned to a model by `maxFrequency`).

The class `Decoder` encapsulates the inverse operations. Its constructor requires an `InputStream` from which coded bits will be read. Two methods make up the programming interface seen by a model:

peek(m) With this method, a model provides its m, describing its reference interval $[0, m]$, and the return value is an integer in that interval. The model must determine which of its subintervals contains the value returned. The corresponding symbol is what the model should return to its caller, but only after calling:

advance(l,u,m) With this method, the model tells the `Decoder` the subinterval that has just been recognized. The `Decoder` scales up that subinterval to become the entire current coder interval, reading more input bits if necessary, in preparation for the next `peek` operation.

During encoding, a model makes just one call per symbol encoded, to the `Encoder`'s `encode` method. During decoding, a model makes two calls per symbol to the underlying `Decoder`. It calls `peek`, determines which subinterval contains the returned value, then calls `advance` to have the `Decoder` "peel off" that subinterval.

The `Decoder` class is abstract. It is implemented by `Decoder32` and `Decoder64`. Like the `Encoder` class, it has a `maxFrequency` method that should be called during initialization of any model that intends to use the `Decoder`.

4.3 Models

The function of a model is to encapsulate the probabilities of the possible instances of a given data type. While encoding, the model should accept as input an instance of that data type, and produce the values l, u, m needed by the arithmetic encoder. While decoding, it should `peek` and `advance` an interval from the arithmetic decoder, and return the corresponding instance of the data type it models.

The flexibility of this coding technique lies in the ability to construct several models that interface to the same underlying arithmetic coder. For example, to encode a record of height, sex, and genome, one would create a model for each of the three data types, connect an instance of each model to a common instance of `Encoder`, then invoke

```
height.encode(h);
sex.encode(s);
genome.encode(g);
```

The genome model in the example is presumably built on a number of simpler models that it uses to encode components of such a complex data type.

In decoding, the same models must be used in the same sequence to retrieve the original values. Note that a good model associates all the valid instances, and only the valid instances, with subintervals that partition its reference interval. *Any* value returned by the coder will fall in one of those subintervals and therefore be decoded as a valid instance of the modeled data type. In fact, as the widths of the subintervals reflect the frequencies of instances, a sequence of arbitrary values from the decoder will be decoded as a statistically plausible sequence of valid instances. If the decoding program rearranges the models by mistake:

```
s = sex.decode();
g = genome.decode();
h = height.decode();
```

it will decode likely sexes and heights, and things that have at least the outward structure of genomes. If it decodes an entire database of individuals this way, it will create a population of bogus individuals whose heights and sexes are distributed just as the model developer expected, but have nothing to do with the values originally encoded.[1] A like result would be observed in "decoding" a random bit stream.

Compression by arithmetic coding effectively reduces redundancy in a data stream, and redundancy is what allows valid data to be distinguished from nonsense. A programmer must ensure that the sequence of models used in encoding is uniquely determined by the input and can be recreated during decoding using only what has already been decoded at each step. More precisely, whatever grammar describes the original data to be encoded, the encoding program must transform it if necessary to encode symbols according to an LL(1) grammar with the added constraint that, at each choice point, all possibilities for the lookahead symbol are encoded using the same model. A simple way to satisfy the constraint is to encode, at each choice point, a small integer explicitly identifying the choice to be taken. The cost of this technique is considered in Section 4.7.

If an encoded stream must be transmitted over a noisy channel, error-correction coding may be advisable; it can be regarded as a deliberate, slight expansion of the compressed data to include an appropriate degree of redundancy.

4.4 Models in Java

Any model is a class that implements the `Model` interface. Such a class encapsulates all the probability information and the machinery to use it as symbols are encountered. A model may also be *adaptive*, updating its internal probability representation to reflect the history of symbols already encountered. Because the state of an adaptive model, on encountering any symbol, is completely determined by the preceding symbols and its initial state, there is no difficulty in decompression using the same adaptive model and the same initial state.

[1] The genomes will also satisfy whatever statistical generalizations might have been built into the model of such a complex data type.

A `Model` itself is not associated with any `Encoder`, `Decoder`, `OutputStream`, or `InputStream`. It has no `encode` or `decode` method. Its constructor accepts only parameters relevant to the probability model itself, such as initial probabilities, set of valid symbols, etc., depending on the model. The `Model` interface requires only two methods of every `Model`:

encoder(e) — Given an arithmetic `Encoder` e (already associated with an `OutputStream`), returns an object implementing the `Model.Encoder` interface, which can be used to encode instances of the data type supported by this `Model`.

decoder(d) — Given an arithmetic `Decoder` d (already associated with an `InputStream`), returns an object implementing the `Model.Decoder` interface, which can be used to decode instances of the modeled data type.

It is intended that every class implementing `Model.Encoder` provide an `encode` method that accepts something to encode as its only parameter(s) and has `void` return type; likewise every class implementing `Model.Decoder` should have a `decode` method of no arguments that returns an instance of whatever data type the model models. The design allows `Model.Encoders` and `Model.Decoders` to be used without obtrusive casts:

```
height.encode(h);
genome.encode(g);
```

where h may be an `int` and g is probably a complicated class. Because different implementations of `Model.Encoder` and `Model.Decoder` are expected to provide `encode` and `decode` methods with different signatures and return types (respectively), the interfaces do not declare the methods. The implementor of a new model must remember to provide them, and may define them with whatever parameters (for `encode`) and return type (for `decode`) will be convenient for the data type being modeled. The design does not eliminate casts from programs that use the models, but it moves them into initialization and eliminates them from the call sites of `encode` and `decode` methods. With some initialization shown, the last example might appear as in Figure 2. Initialization for decoding is similar.

While a `Model` object is independent of any coder or data stream, every `Model.Encoder` and `Model.Decoder` is bound to a coder at the time of its creation. This simplifies `encode` and `decode` methods by eliminating a coder parameter, but there is a more fundamental reason. Before a model can call the `encode` or `decode` methods of a particular coder, it must call the coder's `maxFrequency` method. The model may then have to rescale its frequency data (if it is sophisticated), or throw an exception (if it is not) if the coder precision is less than the model expected. These checks are only done once when the `Model.Encoder` or `Model.Decoder` is created and bound to a coder.

4.5 Basic models

In practice, the `HeightModel` and `GenomeModel` imagined in Figure 2 might be written to implement the `LongModel` and `ObjectModel` interfaces, respectively. These two interfaces describe many of the models one might want to construct. The `encoder` and `decoder` methods of `LongModel` return objects of type `LongModel.Encoder` and `LongModel.Decoder`, and these provide, respectively, an `encode` method with a single `long` parameter, and a `decode` method returning a `long`. `ObjectModel` provides the same support for encoding and decoding the type `Object`.

These are the only two specific model interfaces currently provided. By default integer promotion, the `LongModel` suffices for any integral type, and `ObjectModel` covers everything else but floating point. No

```
import ArithCode.*;
import java.io.OutputStream;

class Example {
  Encoder coder;
  Model hModel, gModel;
  HeightModel.Encoder height;
  GenomeModel.Encoder genome;

  Example(OutputStream o) {
    hModel = new HeightModel();
    gModel = new GenomeModel();

    coder = new Encoder64(o);

    height = (HeightModel.Encoder)hModel.encoder(coder);
    genome = (GenomeModel.Encoder)gModel.encoder(coder);
  }

  void send(int h, Genome g) throws java.io.IOException {
    height.encode(h);
    genome.encode(g);
  }
}
```

Figure 2: Preparing to encode with two models

floating point model has been constructed yet (such fields are rare in audit data) but to do so would not be difficult. Dedicated models for the narrower integral types would contribute to efficiency but add no new functionality.

The `LongModel` interface is implemented by several basic models. The simplest is `EquiprobableLongModel`, whose only two parameters are the endpoints of the allowable value range. Because it assumes all values are equiprobable within that range, it provides no compression other than limiting the number of bits output to the logarithm of the interval width.

Next in sophistication is `FixedLongModel`, whose constructor accepts an array of frequencies for different values. To allow for applications with a wide range of possible values but interesting frequencies for only a small subrange, the array need not be as big as the entire valid range and can correspond to any subrange of it. Outside the subrange mapped to the array, frequencies are taken to be 1.

The `AdaptiveLongModel` is interchangeable with `FixedLongModel` but adjusts its frequency counts during operation to reflect the data stream processed so far. All models support Java object serialization; after training on a set of data, the resulting frequencies can be easily saved and reloaded for later use.

4.6 Composite models

Many audit records contain integer values that are used not to measure but simply to identify. Rather than being smoothly distributed through a range, these fields contain recurring values from a relatively small set. For example, a UNIX[2] system may support 30 000 process IDs or more, but only on the order of a hundred may be in use at any time. For this kind of data, the `DictionaryLongModel` is provided. It assigns nonzero probabilities only to those values that have actually been seen, and only a small "escape" probability for the first appearance of a new value. Probabilities adapt according to the *PPMD* scheme of [6], in which the first occurrence of a new symbol increments both the escape frequency and the frequency of the new symbol itself, each by half the increment used when a known symbol is encoded.

`DictionaryLongModel` is an example of a composite model. Its constructor requires another `LongModel` (of any kind) which will be used to encode the first appearance of any value. Another composite model is `ByteStringModel`, which uses several `LongModels` to encode bytes from the string as well as string lengths falling in different ranges.

4.7 Piecewise models

`ByteStringModel` also illustrates the `PiecewiseLongModel`. This is a composite model whose constructor takes a list of any number of `LongModels`, which may each be optimized for particular sets of values. An exception mechanism allows `PiecewiseLongModel` to send a value to each underlying model in turn, with encoding suppressed, until one is able to represent it. The value is then encoded using that model, prefixed by the encoding of a small integer identifying the model used.

This scheme may resemble the CISL length encoding in Figure 1, but with an important difference. The frequency of use of each submodel is maintained and the identifying prefix is arithmetic coded, as is the model output itself. The conditional entropy of the model output given the prefix is less than the entropy of the original value to be encoded, and the difference is the entropy of the prefix. In this case theory predicts the piecewise implementation will produce an encoding no larger than a single model with corresponding probabilities, and indeed in tests of the `PiecewiseLongModel`, no loss of compression was observed.

The freedom to create piecewise models without sacrificing compression is important for two reasons.

1. In modeling a new data type, a piecewise combination of existing models may be an obvious starting point.

[2]UNIX is a registered trademark of X/Open Company Limited

2. More fundamentally, the fixed precision of the arithmetic coder determines a *maxFrequency* constraint for any model. For example, if `Encoder64` is used, $maxFrequency = 2^{30} - 1$. Any model with an alphabet of n symbols s_i must produce probability estimates $p(s_i)$, scaled into integer values $p'(s_i)$ that satisfy the simultaneous constraints:

$$\begin{aligned} p'(s_i) &\geq 1, 1 \leq i \leq n \\ \sum_{i=1}^{n} p'(s_i) &\leq maxFrequency \end{aligned}$$

   Restricting the $p'(s_i)$ to smaller values restricts the model to coarser estimates of symbol probability; the limiting case $p'(s_i) = 1, 1 \leq i \leq n$ is the equiprobable model. In a single base model there is a clear trade-off between number of distinct symbols representable and the resolution of the probability estimates to be used for their efficient encoding. Even in the equiprobable case, a single base model cannot represent the full range of values of a 32-bit field. Subdividing the range and using piecewise models allows such fields to be encoded.

A piecewise model is also appropriate for a field with a large range of values of a familiar probability distribution. Storing parameters of the distribution is more efficient than storing the frequency of each value. Here handling the distribution piecewise may not only be necessary to satisfy the *maxFrequency* constraint, but may allow the cumulative density, often a computationally expensive function, to be efficiently approximated by polynomials of low degree. Seen in this light, a `PiecewiseLongModel` composed of several `EquiprobableLongModels` is the approximation of some cumulative density function using polynomials of degree zero.

4.8 Parametrized models

In many cases the probabilities assigned to values in one field will vary depending on the observed value of another field. These cases call for a model whose probability assignments depend on a run-time parameter. This can be no more complicated than creating several instances of the same model and selecting one based on some combination of known values believed to affect it. If the model used is an adaptive one, each instance will adapt to the conditional probabilities of transmitted symbols given the conjunction of conditions used to select it. This technique captures the dependency of values such as device numbers on context, such as a disk operation or terminal write. The device numbers that are likely suspects in disk operations are not likely to appear as controlling terminals, and vice versa. To permit decompression, any value encoded using a parametrized model must follow the encodings of values on which its model depends.

One case of a parametrized model that has its own implementation and name is the `PredictedLongModel`. The `encode` and `decode` methods take an additional parameter, a predicted value for the field. The model adaptively encodes a flag indicating whether the transmitted value matches the prediction and then, if not, encodes the value itself using a fallback model. The constructor accepts any `LongModel` to serve as the fallback, and an optional two-element integer array to initialize the relative frequencies of good and bad predictions. The `PredictedLongModel` is useful when knowledge of one item, such as the i-number of a file, predicts with near certainty specific values of other items (such as the file's owner or type), with exceptions only when the i-number is first encountered, after the owner is changed, or after the file is removed and its i-node reused.

4.9 Representing the grammar

The constraints imposed on the input data stream by its grammar are easily seen and exploited when the grammar is expressed formally. In this work, grammars have been expressed in the language of the parser generator JavaCC. Once a JavaCC expression of a grammar is completed, it must be decorated with calls to encode a value at every choice point. At a point with three choices, the value 0, 1, or 2 is encoded using an integer model of three choices. The model can be adaptive and learn the relative frequencies of the three choices, or these can be determined in advance and built into a fixed model. A different model instance may be used at each choice point. For example, the sample productions from an audit log grammar in Figure 3 might be decorated as in Figure 4.

5 A compression tool for BSM audit data

A compression tool for Solaris BSM audit data was built using the framework described here, and starting with a BSM parsing tool, `BSMParser`, developed in an earlier project.[4] The parsing tool comprises a grammar expressed in JavaCC and a class that reads the binary BSM tokens, replacing the JavaCC lexical analyzer. The grammar describes a subset of Solaris 2.6 audit data sufficient to parse our sample audit logs, which include logs created by older Solaris versions as well as recent Solaris 2.6 logs.

5.1 Structure of the compression tool

With the `BSMParser` grammar as a starting point, choice points in the grammar were decorated as described in Section 4.9. In the current implementation, the choice-point actions are the only output-producing actions to appear anywhere but the lowest nonterminals in the parse tree. For example, referring to the parse tree of a **chmod(2)** event in Figure 5, output is produced by a grammar decoration at the *Event* node to record which event production (*chmod*) was recognized. Output is also produced at the *OptAttr* node to indicate that an *attr* node was present. The only other output is produced in the rectangular nodes, which correspond to "tokens" in the BSM file format but to nonterminals in the `BSMParser` grammar.

As each such node is recognized by the parser, its information-carrying subfields are all encoded using suitable models, making use of only the information available at that node, which is then discarded; an abstract syntax tree (AST) is not built. The design is simple, but limiting. For example, it is easy to model the conditional dependence of all of the *subj* fields on *pid*, because all are available in the same node. The models used to encode those fields are adaptive, and reflect the history of values encountered for those fields in *subj* nodes of earlier events. But the current implementation does not exploit the dependence of *error* and *retval* on the comparison of *subj.euid* to *attr.uid*. Likewise, perhaps the condition $subj.ruid = 0$ significantly influences the distribution of likely *path*s, but *path* and *subj* are separate nodes and parsed in the wrong order to exploit the influence. The general solution is to build an AST and do all the output in the *chmod* node, as described in Section 7.

5.2 Modeling BSM

Twenty models have been created for different data types in BSM logs. Three are new models extending `LongModel` or `ObjectModel` and providing their own encode/decode logic. The remainder are trivial extensions of existing models, adding only constructors that pass predetermined parameters to the superclass constructor. Of these, seven extend the `AdaptiveLongModel` and six extend the `DictionaryLongModel`. Four are extensions of `PiecewiseLongModel`.

```
void KillChoice() :
{}
{ LOOKAHEAD( arg() arg() ) KillNonPositivePid()
|                          KillSpecificProcess() }

void KillNonPositivePid() :
{}
{ arg() arg() }

void KillSpecificProcess() :
{}
{ arg() [proc()] }
```

Figure 3: A production for an audit log. One can kill a specific process (identified with a positive PID), or a class of processes (identified with a zero or negative PID). In the specific-process case, the nonterminal proc() describing the target process is optional.

```
void KillChoice() :
{}
{ LOOKAHEAD( arg() arg() ) {T(14);} KillNonPositivePid()
|                          {F(14);} KillSpecificProcess() }

void KillNonPositivePid() :
{}
{ arg() arg() }

void KillSpecificProcess() :
{}
{ arg() ({T(15);} proc()|{F(15);}) }
```

Figure 4: The KillChoice productions after decoration. There are two choice points. T(n) and F(n) are shorthand for “use model n, an adaptive model of two symbols, to encode symbol 1 (or 0, respectively).”

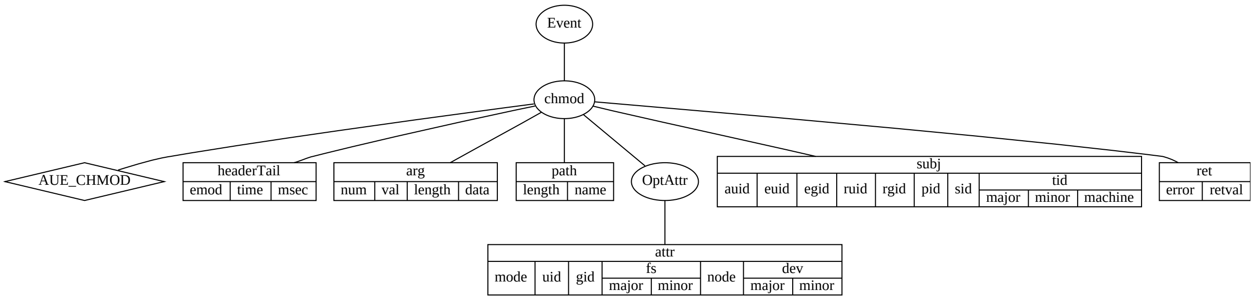

Figure 5: Parse tree of one BSM event.

`DictionaryLongModel` is composite; its constructor requires another model to use whenever a new value must be added to the dictionary. Of the six subclasses of `DictionaryLongModel`, five use simple `EquiprobableLongModels` for that purpose; one uses a `PiecewiseLongModel`. `PiecewiseLongModel` is itself composite; it uses another model for each piece. Most of the `PiecewiseLongModels` used have `EquiprobableLongModels` for all pieces. One uses an `AdaptiveLongModel` for its first piece, and another uses a `DictionaryLongModel` built on an `EquiprobableLongModel`.

The rationale behind the models chosen for various data types can be developed through a few examples. Consider a preorder, left-to-right traversal of Figure 5:

5.2.1 Event

There are 263 possible events. To identify the `chmod` production, an `AdaptiveLongModel` of the integers 0 to 262 is used to encode 34, the position of `chmod` among the event alternatives in the grammar.

5.2.2 EventModifierModel

The *emod* field of the header is a 16-bit field representing an OR of boolean flags. Only 11 flags are defined in `<bsm/audit.h>`, and seven of those concern mandatory access control, which is absent in the base commercial versions of Solaris. Therefore, one may assume the number of possible values is no more than 2^{11} and probably only 16, with some bit patterns very common, others rare. The number of expected values is small enough to choose a `DictionaryLongModel` without fear of memory exhaustion. The fallback is an equiprobable model capable of representing any 16-bit value, so if Solaris uses undocumented flags the field will still be successfully encoded, though memory may become an issue if too many distinct flag combinations are encountered.

5.2.3 MonotoneTimeModel

The *time* in the event header is a 32-bit counter of seconds since Greenwich led the Western world out of the 1960's. Its entropy is much less than 32 bits given that event records are chronological. The `MonotoneTimeModel` adaptively encodes the delta since the previous event if that is less than half an hour; otherwise, it falls back to encoding the absolute time using the generic `TimeModel`. The generic model is coarsely broken piecewise into pre-1995, 1995–2005, and post-2005.

5.2.4 MsecModel

The header field labeled *msec* is documented as a "milliseconds" field, but in the event header its value comes from a Solaris high-resolution timer whose documentation refers to nanoseconds. Observed values of the field can be close to, but never greater than, $10^9 - 1$, and are multiples of 500, supporting the interpretation that the field is a nanosecond counter with 500 ns resolution. An unexpected regularity is observed when the field is printed in decimal: digits 4, 5, and 6 (numbering the units digit 0) have the values 000, 001, 998, or 999 with unusually high frequency. A look at the kernel source would probably explain the phenomenon, but the explanation is not necessary for `MsecModel` to exploit the regularity. The model encodes with adaptive models, in sequence:

- The remainder $msec \bmod 500$, where the value zero is strongly expected.
- The value n where $msec \equiv 500n \pmod{10000}$. There are 20 possibilities.
- The value $\lfloor msec/10^4 \rfloor \bmod 1000$, with high frequencies assigned to 0, 1, 998, and 999.

- $\lfloor msec/10^7 \rfloor$.

It is not uncommon for a series of consecutive event records to have exactly the same value in this field, so `MsecModel` first encodes a flag to indicate whether the value has changed, and omits the four values above if it has not.

5.2.5 ArgValueModel

Of the four fields that make up an *arg*, only *val* contains variable data needing to be saved. The remaining fields identify the argument: *num* gives its position in the system call, *data* (despite the name) is a fixed string such as "new file mode" describing the argument, and *length* is the length of the descriptive string. The descriptive fields are determined by the grammar and position within the production for each event type.[3] Only the *val* field is encoded, usually with the `ArgValueModel`. This piecewise model makes few assumptions about the data, because system call arguments can have such a variety of types. For the range $[0, 4095]$ it uses a dictionary model, on the assumption that such small numbers might be recurring command codes, file descriptors, or the like. The remainder of the 32-bit value space is covered blindly by equiprobable models, but because several such pieces are needed, it is still possible that some compression will be obtained by adaptation of piece frequencies.

A separate instance of `ArgValueModel` is created for each combination of *num* and *data* values, as these indicate arguments with distinct legal value spaces and the distinct model instances are expected to adapt accordingly. An obvious improvement will be to use the *num* and *data* values to select not just a particular instance of `ArgValueModel` but to select a more appropriate class of model. Only one such rule is included in the current implementation: if *data* contains the string "cmd" then a dictionary model with 32 bit equiprobable fallback is selected. Many system calls take an argument labeled "cmd" and the set of command codes is expected to be small enough to keep in a dictionary.

5.2.6 PrefixByteStringModel

The *path* name and its length are encoded together using the `PrefixByteStringModel`. The model first encodes the length of the entire string, then loops to encode segments of the string. For each segment, the model searches its data structure for a remembered string that either is a prefix or shares a prefix. If such a string is found, its index is encoded along with the length of the shared prefix, and the process repeats unless the entire path has been encoded. If, at any point, no match is found, the entire remainder of the input string is sent byte by byte. The data structure is updated by splitting segments or adding new segments as needed. Figure 6 shows a small portion of the model's data structure after encoding a sample of audit data. The model attaches no special significance to the slash character as a delimiter in path names, and will break segments anywhere to factor out a common prefix. This is not unreasonable even for path names, as individual path components often have a common-prefix structure. As it turns out, so do many of the text result strings found in BSM event records, so they are also encoded with their own instance of a `PrefixByteStringModel`. Figure 7 shows that model's data structure after encoding sample data.

5.2.7 Parametrizing models

The models used for the remaining fields in Figure 5 are straightforward, but many of them are parametrized as described in Section 4.8. Usually this is not done by subclassing a model but simply by storing several

[3] There are some BSM events for which Solaris includes *arg*s conditionally, and where two or more *arg*s are consecutive and some are optional, ambiguity results. The solution is for the parser to explicitly consider the values of *num* and *data* when matching *arg* constructs, so the cases are treated as distinct parse trees and the usual encoding at choice points allows unambiguous decoding.

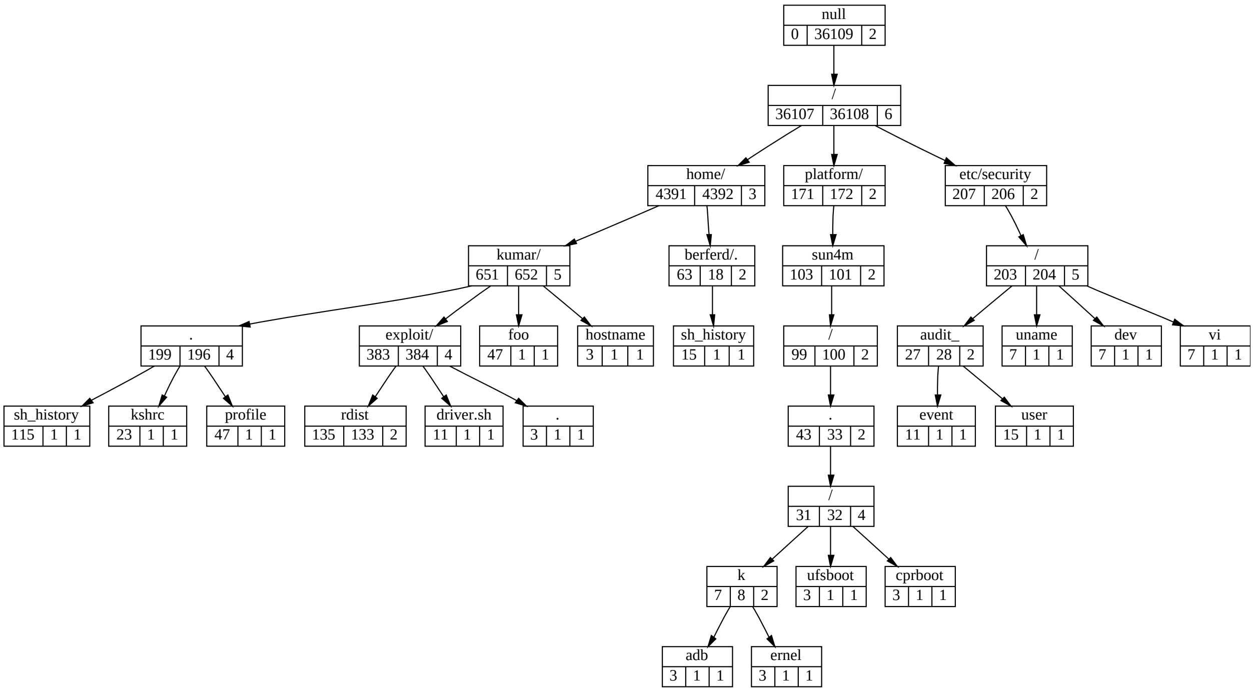

Figure 6: A portion of PrefixByteStringModel's data structure after encoding many path names. The leftmost number at each node is the number of times that node followed its parent. The rightmost number at each node is its *escape frequency*, the number of times a new child had to be added. The middle number is the node's frequency scale m, the sum of the escape frequency and the leftmost values of all the node's children. The numbers in the figure do not add up because many nodes were pruned to fit the illustration on the page.

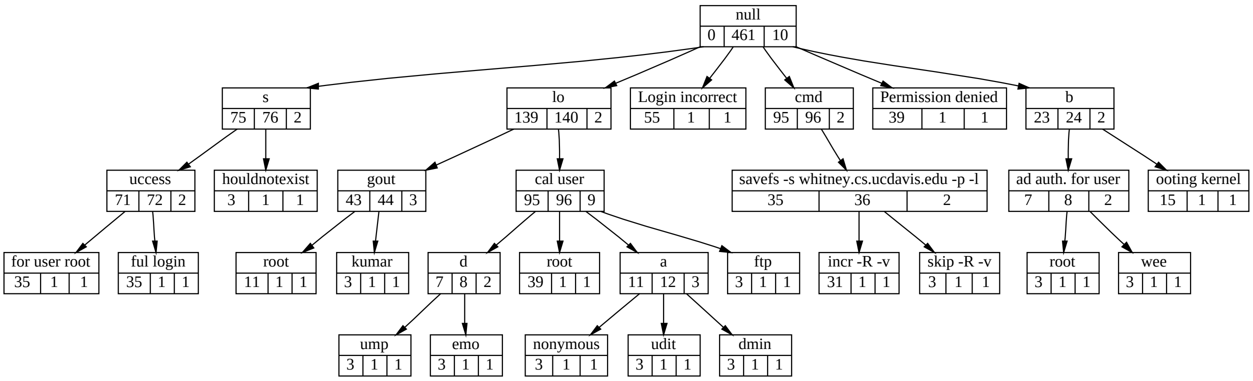

Figure 7: A portion of the data structure after blithely applying PrefixByteStringModel to the text strings in BSM samples.

instances of it in a dictionary, keyed by a short string indicating the context, which may include values of other fields believed to influence the model.

For example, two instances of `PIDModel` are stored, keyed by the strings "pid" and "sid." All session IDs are process IDs, but because the second model instance is used only for SIDs, it grows a smaller dictionary and shorter encodings.

Similarly, `MajorModel` is instantiated for every context in which device numbers can occur: "fsmaj" where a filesystem device is expected, "dvmaj:n" for an arbitrary device, "ttmaj" for a device listed as the controlling terminal for a process. The sets of major numbers learned by "fsmaj" and "ttmaj" can be expected to be disjoint. The n in the string "dvmaj:n" is replaced with the *type* bitfield of the *mode*, so in fact block and character device major numbers have distinct, dedicated instances of `MajorModel`.

The current implementation keys `MinorModel`s by the entire context that applied to the `MajorModel` plus the major number itself. Keying `MinorModel` on major number alone yielded worse compression on our sample data.

Most of the fields associated with *subj* are encoded using an explicitly parametrized model, the `PredictedLongModel`. The predictions are stored in a dictionary indexed by process ID.

5.3 Results

The BSM data to be compressed came from a sample of ten audit logs varying in size from 25 523 to 1 274 868 octets. Three of the logs were created on local equipment and four were obtained from another university; these seven logs were created in 1995. The other three, including the largest of the ten, are recent logs produced by Solaris 2.6 at a Sun Microsystems facility.

5.3.1 Compression ratio

The logs were concatenated to produce a single input data stream of 2 310 114 octets. To establish a baseline for comparison, this data stream was compressed by `gzip`. A parameter, 1–9, can be given to `gzip` to adjust the trade-off of compression ratio against computational effort and memory usage. For a given input, increasing the parameter increases `gzip`'s resource consumption and, for most inputs, reduces the size of the compressed output. The present work involves trading even greater resource consumption for still better compression, so the parameter value 9 was used with `gzip` for the baseline. For the sample input, 9 was indeed the parameter value for which `gzip` produced the smallest output.

The sample input was also compressed with the new tool, twice. For the first pass, all models were initialized with default states: dictionary models with empty dictionaries, adaptive models equiprobable except where reasonable initial probabilities, drawing on domain knowledge, were included in the constructor calls. For the second pass, models were initialized by restoring their saved states from the end of the first pass. The second-pass result suggests the benefit of initializing the models' dictionaries and probability estimates to reflect typical properties of the expected input. That the improvement is modest suggests that minor inaccuracies in the adaptive models' initial states have little adverse effect on compression over an input of reasonable size. That property is important, because any compression advantage from precomputing probabilities for an audit log would be offset by the need to store the computed models for each log compressed. Instead, a facility may compute probabilities once from a representative sample of local logs, and use those stored initial states for all subsequent compression. The compression results are shown in Table 1.

5.3.2 Efficiency

The emphasis of this work has been on developing models with good compression performance without regard to resource consumption. The models, and the arithmetic coding engine itself, have been developed

Method	Output octets	Fraction of input
gzip -9	141 669	6.13%
uninitialized	130 778	5.66%
initialized	122 411	5.30%

Table 1: Compressed output sizes for input stream of 2 310 114 octets

Event span (secs)	Input octets	msec to compress Run 1	Run 2	Run 3
250 250	50 376	3 070	3 121	2 989
1 097 599	137 144	7 931	8 226	7 662
310 647	30 032	1 689	1 733	1 614
28 310	119 897	6 815	6 967	6 585
120	141 332	7 841	8 287	7 894
158 389	26 018	1 506	1 532	1 464
720	25 523	1 366	1 427	1 427
697 633	316 027	17 114	17 843	17 219
86 477	188 897	10 712	10 761	10 408
1 553 220	1 274 868	66 565	66 853	66 205

$$\begin{aligned} msec &= .05197\, octets + 488.8 \\ r &= .9998 \end{aligned}$$

Table 2: Timings for 10 input logs, 3 runs on unloaded SPARC ULTRA 1. Regression suggests about a half second fixed overhead per file and compression rate of about 19 kB of input per second. Event span is the actual interval spanned by the audited events.

in the Java language, making extensive use of standard Java classes such as Hashtable, and without attention to optimization. As discussed in Section 4.1, the coding engine itself does not implement the more recent optimizations to the algorithm. The dictionary models do not yet implement expiration of infrequently used dictionary entries, so memory exhaustion is a possibility, though it has not been observed in these tests. Expiration was omitted only to simplify implementation; in fact, judicious pruning of infrequent dictionary entries should yield improved compression as well as bounded memory use. Finally, this implementation is instrumented to track the number of output bits attributable to each model, requiring evaluation of a transcendental function for every symbol encoded.

The computational cost of the tested compression tool is therefore a very conservative indication of the performance to be expected if these techniques are implemented for production use. Timing results on the sample data are shown in Table 2.

The highest rate of input data production is seen in the fifth file in Table 2, where 141 332 octets of audit data were produced in 120 seconds of high system activity. To keep pace in real time with input data arriving at that rate, the compression tool requires just under 7% of the CPU on a SPARC ULTRA 1. By contrast, `gzip`, with a generic algorithm, native-code implementation, and extensive optimization for speed, is faster by a factor of 34.

5.3.3 Compression results for individual models

Table 3 in Appendix A contains statistics collected for every model used in compressing the sample data. Each time a model instance is used to encode a field, its 'bits in' accumulator is incremented by the size in bits of the field encoded, and its 'bits out' accumulator is increased by the number of bits, possibly fractional, added to the output as reported by the arithmetic coder's `bitsSoFar` method. For example, the `InumModel` was able to encode 8480 i-numbers ($271360 = 8480 \cdot 32$) with 47% compression.

The instances with names beginning 'SYN' are a special case. They are used to encode choices at the choice points in the grammar. Many syntactic features imposed on the input by the BSM specification are distilled by the parser into these encoded choices, but there is no attempt to track the actual number of input bits accounted for by a given choice. Instead, the model's 'bits in' accumulator is increased by the base-2 logarithm of the number of branches at that choice point; that is, the number of bits that would be used if the choice were encoded as efficiently as possible without accounting for unequal frequencies of the branches. Therefore, the 'bits in' column of Table 3 sums to less than the actual size of the input.

The entries in Table 3 are in descending order by 'bits out'; at the top of the list are models where refinement effort might first be invested, either because those models are especially bad, or because the data types they model are especially common in the the audit logs.

Further improvements in compression can be suggested by examining Table 3. For example, several `ArgValueModel`s appear near the top of the list and are seen to have achieved negligible compression. Section 5.2.5 suggested why, and outlined an improvement: use knowledge of specific system calls to determine the data types of their arguments, and use models specific to those data types rather than a single catch-all `ArgValueModel`.

The 47% compression achieved by `InumModel` is respectable, but i-numbers are so common in the input that improvement of this model would be valuable. The present `InumModel` is nothing more than a piecewise model that covers the entire 31-bit i-number range in four pieces. A better idea would be to remember the largest i-number for each file system, and use the file system identification to bound the possibilities for i-number. Probabilities of i-numbers are certainly not independent of the identity and privilege of the subject making the system call, or of the system call being made; a better model would exploit such dependencies. To properly implement such a model, an abstract syntax tree of the entire audited event would be convenient.

As a final example, the present `MsecModel` encodes the fractional-seconds field without considering the whole-seconds value encoded independently by `MonotoneTimeModel`. It assigns a special probability to a fraction equal to the last encoded fraction value. In a better model, that probability would depend on whether the whole-seconds value has changed. Ideally, the whole and fractional parts should be encoded by a single model that properly exploits monotonicity of the entire 64-bit timestamp.

6 Conclusions

The compression achieved by a good generic compression tool such as `gzip` can be surpassed, for richly structured, formally specified input such as an audit log, by building a specialized compression tool with models that reflect the process producing the input. Drawing on domain knowledge, the developer can *select* the models to be used and the data dependencies that will be exploited, but may leave actual probabilities to be adaptively determined in operation. This approach contrasts with techniques developed for general text compression, where the optimal order and structure of the models, as well as the transition probabilities, must be induced from the data. Models deliberately constructed can maintain notions of context much more sophisticated and general than 'the last n symbols,' as richly-structured data may require.

The Java framework described here enabled the rapid development of a compression tool specialized to BSM audit data. Domain knowledge of the Solaris operating system producing the audit data suggested twenty models, whose implementation required only a matter of days using the modeling framework. Over two more days, four rounds of minor refinements to models, guided by per-model statistics as discussed in Section 5.3.3, improved compression performance from 100% worse than `gzip` to 13% better. Table 3 suggests that further significant improvements are possible. Several of the models are still quite coarse and, for the moment, the parser builds no abstract syntax tree, constraining all models to the local notion of context discussed in Section 5.1.

The improved compression comes at a cost. The BSM tool described here is slower than `gzip` by a factor of 34. Although much of that factor reflects the difference between a non-optimized Java prototype and a finely-tuned native code production tool, it is likely that even an optimized arithmetic coder with sophisticated models will incur some performance penalty compared to the generic tool. But for some applications, the penalty may be affordable. Even the prototype would require only 7% of an ULTRASPARC CPU to keep pace in real time with the highest rate of audit data production represented in the sample input.

The arithmetic decoding operation is computationally comparable to encoding. In auditing applications it may be common to compress a log once but to wish to analyze it many times, and faster than real time. Accordingly, for practical application of the techniques presented here, it may be most important to focus optimization effort on the decompression side. One such optimization is to allow indexing into a compressed log and decompressing arbitrary portions, as discussed in Section 7.

7 Future work

7.1 Preprocessor to automate grammar decoration

The BSM compression tool was built by starting with a BSM grammar in the language of a parser generator, and adding by hand the action routines to invoke the proper models as input elements are parsed. This tedious, mechanical process took longer than designing appropriate models, which is the intellectually significant part of the task. The BSM grammar has roughly three times as many productions as the grammar for the programming language C++. The magnitude of that task has so far discouraged experimentation with better models that would exploit context from neighboring nodes of the parse tree. Also, techniques such as naming model instances by short strings, to be looked up at run time, were chosen for simplicity and conciseness in hand coding, at some cost in run-time efficiency.

To be of practical use in prototyping specialized compression tools for other data formats, the toolkit should include a preprocessor that, given a grammar annotated with modeling logic, produces two grammar files containing the needed declarations and action routines to perform compression and decompression, respectively. Optimizations could be performed that have so far been avoided because of the volume of hand coding they would require.

7.2 AST generation and better models

Section 5.1 described the shortcomings of models that can refer only to information in a single node of the parse tree. With the development of the preprocessor suggested above, a more sophisticated tool that produces abstract syntax trees of individual events will allow practical investigation of more sophisticated models.

7.3 Comparison of adaptation strategies

Most models in the current BSM tool start with initial probabilities and continuously adapt as message elements are processed. The technique achieves good compression even if the initial values are quite unrepresentative of the data. A disadvantage is that decompression must start at the beginning to recover any desired part of the log. Fixed models remove that disadvantage, but at a cost in compression ratio. The cost is reduced if the fixed initial values are computed from a good representative sample of input. Compromise adaptation strategies are also possible, and the costs should be explored.

7.4 Random-access decompression

Various methods of resynchronizing the arithmetic decoder and fixed, or suitably-constrained adaptive, models should be explored to allow decompression to begin at points other than the beginning of a compressed stream.

8 Acknowledgements

Portions of this work were supported by contracts MDA904-96-1-0116 and MDA904-97-6-0176 from Maryland Procurement Office, and by sponsors of the COAST laboratory. The persistent questioning of Professor Wojciech Szpankowski improved both my understanding and my articulation of the difference between inducing a model, as in general compression, and crafting a model for input whose structure is already known in detail. Paul Fronberg of Sun Microsystems and Aleph Software Consulting supplied sample audit logs and answered a barrage of questions on the finer points of BSM log format.

A Compression results for individual models

Table 3: Compression results for individual models

Class	Instance	Bits in	Bits out
InumModel	inum	271360.0	143960.02
PIDModel	pid	877728.0	132231.4
MsecModel	emsc	628896.0	103392.88
PrefixByteStringModel	path	288864.0	98206.37
ArgValueModel	arg3:arg	76320.0	69593.555
ArgValueModel	arg2:strioctl:vnode	65664.0	59527.1
AdaptiveLongModel	SYN23	157965.44	58365.94
IPAddrModel	ipaddr	1317600.0	41504.777
ArgValueModel	arg1:addr	37504.0	34015.37
ArgValueModel	arg2:len	38016.0	32202.611
ModeModel	mode	271360.0	29651.875
Dict32Model	retval	628768.0	23755.05
GidModel	filg	271360.0	21396.205
UidModel	filu	271360.0	17023.623
RetErrorModel	reterror	157192.0	10464.3955
Dict32Model	arg2:cmd	82368.0	10250.32
MajorModel	dvmaj:2	49112.0	7310.7783
EventModifierModel	emod	314448.0	7122.6533
MonotoneTimeModel	etim	628896.0	6683.603
ArgValueModel	arg1:fd	79584.0	6593.0205
ArgValueModel	arg4:pri	9760.0	6484.583
MajorModel	fsmaj	118720.0	6175.517
MinorModel	fsmin:32	131526.0	3981.1846
PredictedLongModel	sbsi	628768.0	3144.979
MinorModel	dvmin:2:24	34920.0	2818.4463
AdaptiveLongModel	SYN3	6116.0	2744.531
PrefixByteStringModel	envp	8256.0	2596.2917
ArgValueModel	arg0:child PID	5024.0	2581.406
PredictedLongModel	sbeg	628768.0	2532.0466
AdaptiveLongModel	SYN5	2753.0	2501.0579
PredictedLongModel	sbrg	628768.0	2469.3176

Continued on next page

Table 3: Continued from previous page

Class	Instance	Bits in	Bits out
PredictedLongModel	sbru	628768.0	2352.4873
PredictedLongModel	sbeu	628768.0	2344.316
PredictedLongModel	sbau	628768.0	2274.3606
MinorModel	dvmin:2:41	7848.0	1837.0441
MinorModel	fsmin:172	16290.0	1717.4128
PredictedLongModel	sbmaj	275086.0	1716.9891
AdaptiveLongModel	SYN4	4891.19	1621.0095
AdaptiveLongModel	SYN11	2349.0	1399.175
PredictedLongModel	sbmin:24	337518.0	1343.6332
PrefixByteStringModel	text	3680.0	1013.4064
MsecModel	fmsc	608.0	589.9844
ArgValueModel	arg1:no path: fp	640.0	586.94696
GidModel	preg	248960.0	546.3537
TimeModel	ftim	608.0	539.68097
MajorModel	dvmaj:8	61236.0	511.36987
UidModel	prru	248960.0	509.0477
GidModel	prrg	248960.0	502.33282
PrefixByteStringModel	argv	1664.0	477.69897
ArgValueModel	arg1:base	512.0	470.28885
MinorModel	dvmin:2:42	2592.0	412.88052
AdaptiveLongModel	SYN6	2753.0	383.91995
MinorModel	dvmin:8:0	77778.0	322.89725
UidModel	preu	248960.0	289.43964
AdaptiveLongModel	SYN10	3780.132	285.2558
PrefixByteStringModel	fname	760.0	176.60098
ArgValueModel	arg1:setgroups	1312.0	164.22899
ArgValueModel	arg3:stropen: flag	3264.0	158.0356
MinorModel	fsmin:0	2448.0	132.528
AdaptiveLongModel	SYN1	19663.0	124.34182
ArgValueModel	arg2:level	128.0	121.022934
ArgValueModel	arg1:no path: fd	1440.0	119.174614
AdaptiveLongModel	SYN18	121.0	117.11293
MinorModel	dvmin:2:13	8316.0	116.3309
AdaptiveLongModel	SYN2	31158.336	114.32084
ArgCountModel	argc	608.0	98.647224
AdaptiveLongModel	SYN25	121.0	98.26268
ArgValueModel	arg3:optname	128.0	96.24031
ArgValueModel	arg4:optval	128.0	96.0
AdaptiveLongModel	SYN28	209.0	92.35464
ArgValueModel	arg1:setaudit:machine	96.0	91.2224
ArgValueModel	arg1:setaudit:as_failure	96.0	90.90689
ArgValueModel	arg1:setaudit:port	96.0	90.69849
ArgValueModel	arg1:setaudit:as_success	96.0	90.67807
ArgValueModel	arg1:door ID	248832.0	86.21813
ArgCountModel	envc	608.0	82.48838
ArgValueModel	arg3:new file gid	384.0	78.642365
MinorModel	dvmin:8:1	540.0	74.95452
AdaptiveLongModel	SYN8	152.0	74.28911
Dict16Model	iport	336.0	70.54343
ArgValueModel	arg2:strclose: flag	3584.0	69.2579
AdaptiveLongModel	SYN16	816.0	65.370804
ArgValueModel	arg2:mode	64.0	61.691017
ArgValueModel	arg3:dev	64.0	61.58496
AdaptiveLongModel	SYN17	1885.0	58.924294
AdaptiveLongModel	SYN26	1879.0	58.896675
MinorModel	dvmin:2:11	684.0	58.85909
PIDModel	sid	248960.0	56.886986
UidModel	audu	248960.0	56.886986
AdaptiveLongModel	SYN7	162.0	54.653034
AdaptiveLongModel	SYN38	513.0	53.905052
MajorModel	ttmaj	108920.0	51.563618
AdaptiveLongModel	SYN30	209.0	49.48077
AdaptiveLongModel	SYN32	170.0	47.04406
ArgValueModel	arg1:setaudit:asid	96.0	40.829723
ArgValueModel	arg2:new file mode	288.0	40.740578
AdaptiveLongModel	SYN20	41.0	34.887886
AdaptiveLongModel	SYN9	189.0	33.826977

Continued on next page

Table 3: Continued from previous page

Class	Instance	Bits in	Bits out
ArgValueModel	arg1:uid	512.0	31.27074
PredictedLongModel	sbmin:0	16164.0	30.654287
Dict32Model	arg1:cmd	672.0	27.40481
AdaptiveLongModel	SYN13	33.0	23.0623
AdaptiveLongModel	SYN27	29.0	21.816046
ArgValueModel	arg2:signal	224.0	19.59344
ArgValueModel	arg1:gid	352.0	17.747711
AdaptiveLongModel	SYN19	29.0	17.2761
ArgValueModel	arg2:new file uid	384.0	13.940504
AdaptiveLongModel	SYN36	17.0	11.909039
AdaptiveLongModel	SYN37	506.0	10.980149
ArgValueModel	arg5:flags	576.0	7.8463163
ArgValueModel	arg4:arg	512.0	7.731343
ArgValueModel	arg6:mask	512.0	7.731343
ArgValueModel	arg5:attr	512.0	7.731343
AdaptiveLongModel	SYN15	7.0	7.3297963
AdaptiveLongModel	SYN31	33.0	7.000994
MinorModel	ttmin:24	72.0	6.4229054
ArgValueModel	arg5:optlen	128.0	5.7999754
ArgValueModel	arg1:setaudit:auid	96.0	5.32337
ArgValueModel	arg2:type	64.0	4.681824
ArgValueModel	arg1:domain	64.0	4.681824
ArgValueModel	arg3:protocol	64.0	4.681824
ArgValueModel	arg1:signal	32.0	3.9068906
AdaptiveLongModel	SYN40	3.0	3.129283
AdaptiveLongModel	SYN39	2.0	2.321928
MinorModel	dvmin:8:16122	18.0	1.5849625
MinorModel	dvmin:2:90	36.0	1.2223924
MinorModel	fsmin:173	54.0	1.1375035
MinorModel	dvmin:8:3366	90.0	1.0780026
MinorModel	dvmin:8:16383	108.0	1.0641303
MinorModel	dvmin:2:15	180.0	1.0374746
MinorModel	fsmin:166	198.0	1.0339473
MinorModel	dvmin:8:16126	198.0	1.0339473
MajorModel	dvmaj:1	196.0	1.0264722
MinorModel	dvmin:1:0	252.0	1.0264722
Dict32Model	arg3:cmd	512.0	1.0230836
Dict16Model	socktype	336.0	1.0174875
MinorModel	dvmin:2:21	540.0	1.0121748
MajorModel	dvmaj:10	448.0	1.0114048
MinorModel	dvmin:10:0	576.0	1.0114048
MinorModel	dvmin:2:0	684.0	1.0095862
MinorModel	dvmin:2:22	1044.0	1.006259
MajorModel	dvmaj:13	1652.0	1.0030665
MinorModel	fsmin:171	2124.0	1.0030665
MinorModel	dvmin:13:169	2124.0	1.0030664
MinorModel	dvmin:2:105	6300.0	1.0010316
MajorModel	dvmaj:4	6076.0	1.0008315
MinorModel	dvmin:4:0	7812.0	1.0008315
MinorModel	ttmin:0	139968.0	1.0000492
AdaptiveLongModel	SYN22	24.0	0.9708536
AdaptiveLongModel	SYN29	19.0	0.96347404
AdaptiveLongModel	SYN21	11.0	0.9385994
AdaptiveLongModel	SYN0	10.0	0.9328858
AdaptiveLongModel	SYN24	7.0	0.9068905
AdaptiveLongModel	SYN14	7.0	0.9068905
AdaptiveLongModel	SYN35	3.0	0.8073549
AdaptiveLongModel	SYN12	3.0	0.8073549
AdaptiveLongModel	SYN34	1.0	0.5849625
AdaptiveLongModel	SYN33	1.0	0.5849625

References

[1] L. Peter Deutsch. *RFC 1951 DEFLATE Compressed Data Format Specification version 1.3.* Network Working Group, May 1996.

[2] L. Peter Deutsch. *RFC 1952 GZIP file format specification version 4.3.* Network Working Group, May 1996.

[3] Rich Feiertag, Cliff Kahn, Phil Porras, Dan Schnackenberg, and Stuart Staniford-Chen. A common intrusion specification language. http://gost.isi.edu/projects/crisis/cidf/cisl_current.txt, December 1998.

[4] Chapman Flack. Formal specification of audit log content using machine readable grammars. Technical Report in preparation, Purdue University, 1999.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[6] Paul G. Howard and Jeffrey Scott Vitter. Practical implementations of arithmetic coding. Technical Report CS-91-45, Brown University, July 1991.

[7] Gordon Irlam. Unix file size survey-1993. http://www.base.com/gordoni/ufs93.html, September 1994.

[8] Sun Microsystems. Java Compiler Compiler (JavaCC) - The Java Parser Generator. http://www.suntest.com/JavaCC/, May 1997.

[9] Sun Microsystems. *SunSHIELD Basic Security Module Guide.* Sun Microsystems, 901 San Antonio Road, Palo Alto, California, solaris 2.6 edition, 1997. Part Number 802-5757-10.

[10] Abdelaziz Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection.* D.Sc. thesis, Universitaires Notre-Dame de la Paix Namur (Belgium), September 1997.

[11] Kihong Park, Gitae Kim, and Mark Crovella. On the relationship between file sizes, transport protocols, and self-similar network traffic. In *Proc. International Conference on Network Protocols*, pages 171–180, October 1996.

[12] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 623–656, July 1948.

[13] Stuart Staniford-Chen. The DARPA common intrusion detection framework. http://seclab.cs.ucdavis.edu/cidf/, December 1998.

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