Deciphering the SAS file format

Gene Michael Stover

created Sunday, 2007 March 11
updated Sunday, 2007 April 15

Contents

1 What is this?

My brother works on the Gnu PSPP project, & he asked me to figure out the format of data files for SAS, a competing application.

Here's how I did it, complete with source code for programs I wrote while I deciphered the file format.

1.1 Automated deciphering

If I do most of the deciphering work in my own head, then I have to document my technique & my calculations. Then if we ever need to perform the deciphering again, someone must update the ideas, then calculate again, & success will depend on the mind of the person doing the calculations.

if I write a program which does the calculations & produces the answer, then I still have to document my technique & how the program works, but

the program itself documents the process, & the program in combination with the input files can reproduce the calculations. If we ever need to improve the technique, we improve the program & run it. What's more, success at run-time does not depend on the mind of the person running the program. Success still depends on the minds which created the program, but it seems less dependant than if a human performs all the calculations.

What's more, if I write a program which does the deciphering, then anyone can verify that the program works by running it. We can verify its answer by running it on other input data sets.

1.2 World without boundaries – a plague of deciphered formats

I hope that lots of people write lots of programs which automatically decipher lots of file formats. It could help remove the obstacles created by undocumented file formats.

Files in formats which are documented are more useful because programmers can write their own programs to process the data. If the only way to access a file is through a single application, then if someone wants to perform their own calculations on the data, the application's creator must add hooks for general-purpose processing. This bloats the application, makes it less reliable, more expensive, & less fun to use. Jeez, it just plain sucks. But if other programs can use the files, then the original application can remain minimalist & do what it does best.

If it became common for people to write programs which decoded file formats, a lot of file formats would cease to be undocumented.

Yep, some evil entities would respond by encrypting their files, but that extreme gesture might help more people wake-up to the costs of undocumented file formats. And who knows? Some clever programmer of free software[1] might even decode the new file format in spite of the encryption.

2 Disclaimer

I'm no cryptanalyst, so even though this is a simple problem in cryptanalytical terms (a chosen-plaintext attack in which all messages are enciphered with the same key), I'm sure my techniques are less than optimal. I don't apologize for this.

[1] "Free" as in Freedom, not as in beer.

3 The plaintext & encoded files

4 Log 0

1. Use "`(cd files-2; ./sas-lisp-generator >manifest.lisp)`" to create `manifest.lisp`, a description of the data sets in Lisp.

2. The `files-2/manifest.lisp` file has nine fields. They are ...

   n vars the number of variables. Always 1 or 2.

   var 1 value the value of the first variable. If the type of FLOAT, this is a number. If the type of UC or LC, this is a string.

   var 1 type the type of the first variable. Always one of the symbols FLOAT, UC, or LC.

   var 2 value If there is only one variable, this is NIL. Otherwise, this is the value of the second variable: a number or a string.

   var 2 type If there is a second variable, this is its type, one of the symbols FLOAT, UC, or LC. If there is only one variable, this is still one of those symbols (usually/always FLOAT).

   data name base filename of the SAS data file. To get a pathname, you need to remember that it's in the "`files-2`" directory & that the file type is "`sas7bdat`".

   var 1 name the name of the first variable, always a string.

   var 2 name If there are two variables, this is the name of the second, always a string. If there is only one variable, this is NIL.

3. In Lisp, make a structure type for convenient access to the manifest records ...

```
(defstruct manifest
  (pn nil :type pathname)
  ;; (n-vals 1 :type (integer 1))
  (n-vars 1 :type (integer 1))
  ;; var-1-type WE DON'T USE THIS FIELD
  var-1-value
  ;; var-2-type WE DON'T USE THIS FIELD
  var-2-value
  (var-1-name "" :type string)
  var-2-name                          ; a string or NIL
  sas)                                ; octets from SAS file
```

4. In Lisp, make a function to load the manifest:

```
(defvar *default-manifest-pathname*
        (make-pathname :directory '(:relative "files-2")
                       :name "manifest" :type "lisp"))

(defun read-manifest (strm)
  "Return the next MANIFEST from the input stream
or NIL."
  (declare (type stream strm))
  (assert (input-stream-p strm))
  (assert (open-stream-p strm))
  (assert (eq 'character (stream-element-type strm)))
  (let ((x (read strm nil)))
    (if x
(let ((pn (make-pathname :directory '(:relative "files-2")
 :name (string-downcase (sixth x))
                                  :type "sas7bdat")))
  (make-manifest
   :pn pn
   :n-vars (first x)
   :var-1-value (if (equal "float" (third x))
    (read-from-string (second x))
  (second x))
   :var-2-value (cond ((equal "NONE" (fourth x))
                               ;; There is no second variable;
                               ;; there's just one variable.
                               nil)
                              ((equal "float" (fifth x))
       (read-from-string (fourth x)))
                              (t (fourth x)))
   :var-1-name (seventh x)
   :var-2-name (eighth x)
   ;; The manifest file isn't entirely
   ;; accurate.  Some of the pathnames
   ;; are krap.  The easiest way to deal
   ;; with that is to ignore it.  It's
   ;; one of the gillions of examples of
   ;; how the person who creates the data
   ;; file must be given an automaton which
   ;; consumes the data so the person can
   ;; tell when the data file is correct.
   :sas (with-open-file (sas pn :element-type '(unsigned-byte 8))
  (coerce
   (loop for y = (read-byte sas nil)
 until (null y)
 collect y)
   '(array (unsigned-byte 8) (*))))))
```

```
  ;; else, It's end of file.
  nil)))

(defun load-manifest (&optional (pn *default-manifest-pathname*))
  (with-open-file (strm pn)
    (loop for x = (read-manifest strm)
          until (null x)
          collect x)))
```

5. Load the manifest into a global ...

```
lisp> (defvar *manifest* (load-manifest))

*MANIFEST*
lisp> (length *manifest*)

844
```

6. Verify that all of the manifest entries have SAS files. In other words, verify that every manifest entry has a vector of octets.

```
lisp> (count nil *manifest* :key #'manifest-sas)

0
```

   Good.

7. The basic idea is to pick a technique for encoding a type, then count the cases in which the technique produces a sequence which is in the SAS file for that case. The higher the count for a technique, the more likely that technique is used in the SAS file.

```
(defun count-technique (technique test key &optional (lst *manifest*))
  (declare (type function technique test key)
           (type sequence lst))
  (count-if #'(lambda (m)
                (declare (type manifest m))
                (let ((value (funcall key m)))
                  (and (funcall test value)
                       (search (funcall technique value)
                               (manifest-sas m)))))
            lst))
```

8. For floating point numbers, let's start with 32-bit IEEE encoding in big-endian order. Function ENCODE-IEEE-SINGLE-FLOAT in package COM.CYBERTIGGYR.GENE.FF0 in file "SRC/FF0.LISP" does that.

```
lisp> (load "src/loadall.lisp")

T
lisp> (import 'com.cybertiggyr.gene.ff0:encode-ieee-single-float)

T
lisp> (count-technique #'encode-ieee-single-float
                       #'numberp
                       #'manifest-var-1-value)

170
```

9. Let's try the same technique but little-endian.

```
lisp> (count-technique #'(lambda (x)
                           (reverse (encode-ieee-single-float x)))
                       #'numberp
                       #'manifest-var-1-value)

175
```

5 How could SAS defeat these efforts

What if the makers of SAS don't want me to be able to decipher their file format? It's too late for SAS version 9, but they could change the file format for SAS version 10.

The easiest modification might be to retain the current file format but encrypt the bits before emitting them to external storage. They wouldn't even need the latest cryptosystem. Plain old DES is beyond me & most programmers. Yeah, yeah, the NSA might be able to defeat DES in a single day, but if the NSA (acting on behalf of the federal government of the USA) wants to read your secret data[2] or determine your secret file format, you have bigger problems than clever cryptanalysts. Think money, lawyers, & prison. And there's no reason SAS couldn't use the latest, woopie-doo cryptosystem of the month. I just said that something older would do the trick just as well.

With a little more effort, SAS's file format could be protected with less security. Here are some possibilities:

- Compress the files with any of the standard compression techniques.
- Forget about octet boundaries for fields. Use bit boundaries as does ASN.1's Packed Encoding Rules (PER).
- Don't store all integers as fixed-length fields. Store them as offsets from a base as does ASN.1's Packed Encoding Rules (PER).

[2]Is it really secret? Is it secret for a good reason?

- Use variable-length fields (length, type, value) as does ASN.1's Basic Encoding Rules (BER).
- Use their own external character encoding so that when someone dumps the octets from the files, they don't see readable strings.
- Store fields out of their obvious, logical structure. Use a minimum sized index elsewhere in the file. This technique might also help reduce the file's size.
- If they have the option of row-major versus column-major order, dynamically switch between them, using another field in the file to indicate to the parser which technique is used. This could also apply to endianness, strings (length-prefixed or zero-terminated), & probably to other decisions. It can be applied to the file as a whole or to each field.

A non-cryptanalyst programmer (such as I) could defeate any of the preceeding techniques, but they would slow him & the deciphering program he wrote. It's like I said: "With a little more effort, SAS's file format could be protected with less security" than actually encrypting it.

On the other hand, if the makers of SAS wanted to make it even easier for other programmers to write programs which processed SAS files (thereby increasing the utility of SAS itself), they could document the file format. Here are some specific things they could do:

- Document the format in a natural language. Publish or post the document ont he web, maybe even as an Internet RFC.
- Document the format in a machine-processable way such as ASN.1 (which I prefer) or XML Schema. Combined with semantic notes in a natural language, this might be the ultimate way of documenting the file format because other programmers could feed the specification to a protocol compiler to obtain I/O code in their language of choice. Any programmer could process the files with any language he chose. Nice.
- In liew of machine-processable documentation such as ASN.1, use a text-based encoding such as XML, CSV, INI, or Lisp (whichever better fits the data's structure). Even a proprietary text-encoded language would be okay.

A Other File Formats

- This document is available in multi-file HTML format at http://cybertiggyr.com/gene/ff0/.
- This document is available in Pointless Document Format (PDF) at http://cybertiggyr.com/gene/ff0/ff0.pdf.

References