Notes on the Transclusion Finder

The html version of this document badly botches the special symbols. The .pdf or .ps versions are good in this regard. The text with the formulae may be of interest to some however.

The main idea is to define a "landmark" that is a property of a string of bits that is suitably rare. Each file is scanned for landmarks and a checksum of the data around each landmark is made and included in a file along with the address of where it was found. The file is them sorted by checksum and then landmark sites with the same checksum are compared by reading the disk.

The main difficulty is finding a suitable landmark definition. I invented a stream hash function that is fairly efficient to compute. It amounts to a convolution of the bit stream of the file with a string of about 500 bits. I look for strings of about 11 ones in the hash and call this a landmark. Bits surrounding this string of ones are used as the checksum. This has the convenient property that if two file portions are alike except for different offsets then the coincidence will be found even if the offset difference is not a multiple of one byte.

This scheme is likely to find coincidences of more than 256 characters as such a stream is likely to have a landmark. Shorter coincidences are less likely to be found.

One piece of hair is removal of multiple reports resulting from multiple landmarks in one coincidence.

Local Stationary Bit Stream Functions

This is the theory of a string hash function. It is a function on a long string of bits. Its value is a few hundred bits longer than the argument. It is local, stationary and fairly fast to compute. We use “string” here to refer to arrays of bits indexed by the integers. If b is such a string then b_j is bit j of that string. The strings are infinitely long in both directions but the strings will be zero outside some finite range. Both the argument and value of string hash is a string. b << n is the string b shifted n to the left. $(b<<n)_i = b_{i-n}$ for any integers i and n. Note that we picture bits with a larger index on the left! e.g. $b_2\ b_1\ b_0\ b_{-1}\ b_{-2}$ ... —this is the classic order for polynomial coefficients. We also write “b(x)” to represent the polynomial $\Sigma b_i x^i$ over the field GF(2).

A string function s is stationary iff s(b<<1) = s(b)<<1 for all b. In this case s(b<<n) = s(b)<<n for all b and n.

A function s is local when there are integers a and b such that a<b and $s(b)_i$ is determined by b_j only when $i+a < j < i+b$. This means that such functions depends on only bits in the neighborhood. More precisely L is local if there are integers a and b such that a<b and for any strings s and t (for all i, if (for all j if $(i+a < j < i+b)$ then $s_j = t_j$) then $L(s)_i = L(t)_i$).

We want a stationary function so that properties of a string that are preserved under shifting will remain visible in the yield of S if they were visible before. We want the string to local so that equality of strings of modest length will lead to equality of the yield of S. We will look for “landmarks” in the yield such as strings of ones. No particular pattern can be presumed to have a useful density in the original data—zeros, for instance, are excluded from ascii text. Such patterns should have known densities in the “randomized” data that comes from S.

Our particular string hash function is based on a polynomial h over GF(2) of degree $P \cong 50$. We identify a bit string b with the polynomial $h(x) = \Sigma b_i x^i$ summed over non-negative i.

When polynomials are used to checksum transmission blocks a “checksum” is computed and transmitted that would have caused the block to be a multiple of the polynomial had it been exclusive-ored into the right end of the block. In this case the checksum is a function of all of the bits in the block and is thus not local.

Our string hash function S is a convolution function. $S(b)_i = \Sigma b_{i-j} p_j$. (In this context Σ means the one bit exclusive-or: ⊕) If $h(x) = x^{64}+x^{61}+1$ then p may be recursively defined as $p_j = 0$ for $j<0$ or $j \geq B$ and $p_0 = 1$ and $p_j = p_{j-64} \oplus p_{j-61}$ for $0 < j < B$. (B is some constant $\cong 500$). Alternatively p may be defined by the relations $x^B + r(x) = p(x)h(x)$ and $\deg(r) \leq \deg(h)$. r(x) is the remainder after dividing x^B by h(x) and $p(x) = (x^B + r(x))/h(x)$. It may be seen that S is stationary and local for the range $a = -B$, $b = 0$.

The “polynomial” associated with p for infinite B is the reciprocal of h in the sense that the terms of the product of p and h are the coefficients of the polynomial 1 despite the fact there are infinitely many terms in p. This is like $1/(1–x) = 1+x^{-1}+x^{-2}+x^{-3}$... You may have seen the old fallacious proof that 1+2+4+8... = -1. That proof is valid in this funny world.

The desired convolution is m(x)p(x) where m_i are the bits of the argument to S. If there are N bits in the argument then the first bit is m_{N-1} and the last bit is m_0. We will process the bits from first to last.

$m(x)p(x) = m(x)[(x^B + r(x))/h(x)] = [m(x)(x^B + r(x))]/h(x)$

We first describe a conceptual method that does not have the factor B in its cost. Then we show how to do it several bits at a time instead of one. The first conceptual method of computation proceeds as follows. The string argument is delivered one bit at a time, largest index first. The algorithm does not know the value of that index. S being stationary, we can compute the answer as soon as the data begin to arrive. By the first and last bits of the string argument we refer to some range of the infinite input outside of which is all zero. We buy three left infinite accumulators m, s and q. m accumulates the argument, s will accumulate $m(x)(x^B + r(x))$ and q will accumulate the quotient $[m(x)(x^B + r(x))]/h(x)$. We prepare the initial answer as if the input string were empty (all zero) by setting m, s and q to all zeros.

When ever a bit of m arrives:

1) Shift m, s and q each to the left one bit and deposit the new bit in m_0.
2) If the new bit is 1 we exclusive-or $x^B + r(x)$ into s.
3) Set q_0 to s_B.
4) If s_B then exclusive-or $h(x)x^{(B–P)}$ into s. (This turns s_B off.)

When an oracle tells us that there are no new bits we supply B–P zeros after which s will be all zeros. Note that the last step above keeps $s_j = 0$ for j>B. Supplying zeros continues to increase the number of low bits in s.

The loop invariant for the above is:

$s(x) = (x^B + r(x))m(x) – q(x)h(x)x^{(B–P)}$ and deg(s)<B.

Since s is driven to zero we have $q(x) = (x^B + r(x))m(x)x^{(B–P)}/(h(x)x^{(B–P)})$ upon exit from the loop. The extra factor $x^{(B–P)}$ in $m(x)x^{(B–P)}$ above arose when we fed B–P more zero bits into the algorithm after the last bit of the argument arrived.

Steps 1 and 2 constitute the multiplication $(x^B + r(x))m(x)$ and steps 3 and 4 perform the division $(x^B + r(x))m(x)/h(x)$. We now have:

$q(x) = (x^B + r(x))m(x)/h(x) = m(x)p(x)$

which is the desired answer.

....We reformulate the above in notation without mutation such as shifting and “exclusive-or”ing.

To accelerate the above computations we consider the data flow. The only parts of the computation that are proportional to B are the shifts of s, m and q. m is entirely unneeded—it was computed only to make the the loop invariant testable and thus make the computation easier to debug. The bits of q are produced as the answer and are unused again in this computation. Only the first B bits of s are used—indeed the rest are all zero. After s_P is produced it is not used again, aside for shifting until it becomes $s_{B–P}$. This is thus a fixed length FIFO queue and may be implemented without shifting. Alternatively we can dispense with the queue and perform steps 3 and 4 sooner if we can get an early edition of the input bits so as to perform the x^B element of step 2. The latter is the only thing delaying the computation in steps 3 and 4. In either case there remain no factors of B in the computation.

Substantial further savings are possible by processing several bits of input at a time. For simplicity we will describe the 8 bit strategy. As the algorithm processes 8 bits, its storage, s, undergoes a linear transformations. s_0 thru s_{P-1} are transformed by being shifted 8 to the left, becoming s_0 thru s_{P+7}, and having a P+8 bit string depending on the next 8 input bits, exclusive-ored into them. The P+8 bits may be found in a table of pre-computed constants indexed by the next input byte.

Bits $s_{B–P}$ thru $s_{B–1}$ undergo a similar linear transformation. Any time after the input bits have been contributed at position x^B eight iterations of step 4 may be done in an 8 bit left shift followed by an exclusive-or of P bits into s_{B-P} thru s_{B-1}. These P bits are taken from another constant table indexed by the old bits s_{B-8} thru s_{B-1}. Yet another table indexed the same yields the 8 answer bits.

There is a simple but unsatisfying proof of the above. The output is a linear function of the input in the sense that input and output are in a vector space over GF(2). The answer is correct when the message has a single one bit. (you can work this out in your head.) It is therefore correct for any argument with a single bit since S is stationary. The messages of a single bit form a basis for the vector space. The algorithm above computes a linear function. Linear functions that agree for each member of a basis agree everywhere. Q.E.D.

Some Combinatorics

How many bit strings of length n have at least one contiguous string of eight ones? A recursive formula is $S_i = 2S_{i-1} + (2^{i-9} - S_{i-9})$, $S_8 = 1$ and $S_0 \ldots S_7 = 0$. The idea behind this formula is that an n bit string is composed of the concatenation of a one bit string and an n-1 bit string. We use “8string” to designate a bit string with one or more runs of 8 ones. Each n bit 8string is in exactly one of the following three forms: a zero followed by an n-1 bit 8string, a one followed by an n-1 bit 8string or a one followed by an n-1 bit string that begins with 11111110 and is then followed by an n-9 bit string that is not an 8string. The last proviso avoids double counting. The formula above follows from this trichotomy.

This yields $S_{32} = 131730944$.

Running the Program

To invoke TF:

```
TF [-s options] directory_name.
```

"options" is a string of letters, each suppressing a different form of diagnostic output to "terminal". The more characters the less output. Without the -s option, "`-s CL`" is assumed. "`directory_name`" is traversed recursively.

Classes of diagnostic and amusement output:

"F" names each file as it is first read, and directory as it is entered.

"R" reports how many characters in each file and prints "b" for each block.

"L" reports details about where matches were found.

"V" is for vernier faults, unfortunate and rare.

High Level View of Program

The above stream hash function has a mathematical style of theory. The rest of the program is anything but mathematical.

Indeed this is an ugly program. It would be rather improved if written in C++, but still ugly. Part of the ugliness is due to the lack of a crisp spec.

There are two phases to the program and it is currently coded to finish one phase before starting the next. There is some flexibility to merge the phases and achieve trade-offs for space and against execution time.

Phase one reads the accessible files, eliminating duplication arising from links. The large output of this phase is a list of marks. Each mark is represented by an instance of the `sx` structure that comprises a 32 bit signature and a 32 bit mark locator. This is accumulated into a large array `LM`. A small output of phase one is a coded map which can translate a mark locator into a file name and a block within file.

At the end of phase one we sort the marks by signature and carry along the locations. Then in phase two we pass over the sorted list. When we come to a pair of adjacent equal marks we call `locate` and read the files where the marks were found. We extend the equality span as far as possible. Then, so as not to process this span again as triggered by other marks in the same span, we remove the marks from the sorted list associated with one of the two files. We do this by again passing over that file portion and for each mark find we do a binary search in the sorted mark list and “remove” the mark from the list. We cannot merely zap the entry for that would break the binary search code. Instead we disable the entry by changing its location field to -5. We keep such invalidated mark entries at the end of the run of entries with equal signatures.

Navigating Unix File Systems

Directory and file links present a special hazard for our purposes. Loops in directories can prevent termination and even without loops would cause false transclusion indications. The `lstat` system call in SVR4 alerts us that a directory entry is a symbolic link. We merely ignore such entries. Hard links to files are a bigger problem as the entries are symmetric. `lstat` returns a structure that includes a count of hard links to the file. To avoid double counting we remember the “i_node” number of each file with more than one hard link and visit the file only the first time we find a directory entry (link) to it.

Locating Data in Files

We must remember where each landmark was found. Considering that we must remember and sort millions of landmarks we must code this memory efficiently. Since S is strictly local it may be quickly recomputed starting anywhere in a file. We choose a block count to locate both which file and where in the file. A block is 512 bytes even where the underlying file system block is something different. This provides a 32 bit locater which is unique within two terabytes of storage.

We imagine the accessible files to be concatenated in directory order and their blocks numbered in that concatenation. This means that we must provide an auxiliary memory of which file holds block numbers within a given range. This memory will be large but not nearly so large as the landmark memory. I propose that we form a chronological and linear record of each directory and file visited on the first pass. If we merely record the size of each file then the file may be efficiently located. Each directory entry will record the total size of the files within and point to the previous directory entry at the same level. This allows rapid location of the file given the block locator. It is also used in conjunction with the `opendir` call to reconstruct the file name that is necessary to open the file to confirm the transclusion. This form of file memory is suitable to writing into a file during the first pass and landmark sort. The pointers between sibling directories are relative to the file memory.

We build a large FILO structure FM to hold our file memory. We build it in one order and consult it repeatedly in the other order. Abstractly FM is a stack of integers—it grows like a stack but does not shrink as it is used backwards. As we process a file yield of `readdir` we append something to FM. If the yield indicated a file we append the file’s block count. If we cannot process the file, perhaps because access was denied, we append zero. If the yield indicates a directory we note the FM cursor and the current global block count. We then process the entries in the new directory. After we exhaust that directory we append the size of the data appended to FM for this directory. We also append the total block count. We then a append -1 as a distinctive marker indicating a directory. This allows to skip this part of FM as we read FM. Since FM is large and most of its integers are small we compress them by techniques best understood by reading the code.

There is an infelicity in this design. It takes special logic to discover a string that is repeated in the same file block. We omit that logic for now.

Some Program Logic

`Lmain.c` and **`Umain.c`**

are alternative files. `Umain.c` is the production version and `Lmain.c` is a test version.

First Phase: Scan the Files

`rd` is a recursive routine that reverses the directory structure. It has the logic to avoid loops. `rd` opens each leaf file and calls `fl` passing the open file and its length.

`fl` sets **`rlm`** so that each encountered landmark calls **`Rlm.`**

`fport`

`fport` is a data structure to designate a file and a point in that file. These are short lived, on the stack, and do not persist between the two phases.

`sx`

`sx` is the structure that remembers the location of a landmark between phases. There is the array **`LM`** of `sx`'s.

`locate` is the only code that makes values for `fport`s. It opens files in phase two.

`scan`

`scan` proceeds forwards thru the stream hash and watching for landmarks. `scan` calls `so` for each anded (see `so`) word. `scan` is shared between the two phases. It is called by `fl` for each file in the first phase and `vernier` and `forget` during second phase.

`Much` is a variable global to file fl.c which is the size in words of the file being read.

`gaw`

`scan` calls `gaw` the for each word of hashed stream. `gaw` calls the system fread function to get data from the file.

`vernier(fport * C p, b32 C bn, fport C * C dp)`

`vernier` starts with a crude recollection of where a landmark was found in the first pass and locates it exactly. It does this by rereading and computing the string hash.

`void find_match()`Scans the list of landmarks, sorted by signature, looking for the same signature at different points. When it finds the same signature at for two places it calls `span`.

`span(fport C * C X, fport C * C Y, sstream * C x, sstream * C y)`is local to file p.c. It is called when two marks with the same signature are noted. Its job is to maximally extend the transclusion forward and backward. **`X`** and **`Y`** identify the marks and where they were found. The bit offset for **`X`** is ≤ that of **`Y`**. This is significant only when the files are the same. It is easy for `span`'s caller and convenient for `span`. x and y are open sstreams for the respective files available for accessing the files. **`span`** makes the transclusion report and removes (calling `forget`) the marks from the base generated by the first file in the specified span.

`sread(uchar * wh, sstream * s, b32 C loc)`

`sread` is like `fread` except for having improved read backwards performance, taking a modified stream parameter and doing bit oriented offsets. `sread(uchar * wh, sstream * ss, b32 loc)` reads 512 bytes from stream `ss` at bit offset `loc` into `wh`. The `sstream` thing buffers some data to improve performance. `sread` uses the buffer in the sstream as a cache for this file.

`void sopen(sstream * s, FILE *f)` opens an `sstream` on an already open file.

`sstream`

This is a structure that lives on the stack in phase two. It includes a file descriptor and 1K buffer.

`void forget(fport C * p, b32 start, b32 stop)` locates and removes all marks from the base that were generated by **`p`**'s file between byte offsets **`start`** and **`stop`**. It does this by re-doing the hash function over that span and doing a binary search in the base for each mark regenerated. Marks are removed only if the `loc` field in the mark indicates it came from the start-stop span since the same signature may appear elsewhere in the file and thus in the mark data base. Marks are "removed" by changing their `loc`'s to -5. Modifying their `sig` field would break the logic of binary search! Some future version of this program might learn to consolidate the mark data base by removing the dead entries and then shrinking it. (Note the peculiar effects of the signedness coercions of the arguments of the `max` function. `max` operates on signed values so as to make otherwise unsigned values such as ~0 seem small.)

The routines **`w_log`** and **`r_log`** store and retrieve a list if 32 bit integers. They compact leading zeros in these integers. `r_log` retrieves successive integers in the reverse order that `w_log` stored them. **`log_crsr`** is a variable that can be read and written to provide random access to the integer store. `w_log` and `r_log` are used to record an excessively clever data base called name-memory and located by **`log_crsr`** that is used to convert a global block number recorded in that mark data base back into a file name. It is very compact but a bit delicate. The delicacy is that it may fail if a directory in the path for the file name is changed between forming the memory (first phase) and consulting it (second phase). I don't know a way overcome this delicacy short of vast increase in the size of the main mark data base. It could be made more self diagnostic by including some sort of name hash which could be used to detect most of these changes. This would double the size but not ultimately improve the situation for there would be nothing safe to do to find the real old file. The logic of the name-memory assumes constant directories. For each item in a directory (file or sub-directory) it remembers the number of global blocks traversed while processing that entry in phase one. When phase one finishes a directory It assumes that the order and number of the items in a directory as presented in the SVR4 system call `readdir` is the same in phase one and two. When this fails transclusions are lost and copious diagnostic messages arise unless suppressed. No other ill effects should result.

`so` is called once for each word of the "anded stream hash". This stream is the and of eight streams consisting of the stream hash shifted i bits to the right for i from 0 thru 7. A one bit in this stream indicates eight consecutive ones in the stream hash and thus a mark. In order of "early to late" `so` finds each 1 bit and its offset in the word.

It calls a deeper routine with the bit offset within file and 32 bits of relevant bits of stream hash computed from array **`grunge`**, a small circular array or stream hash bits. These hash bits are aligned with the landmark.

Which deeper routine it calls, **`Rlm`**, **`sm`** or **`forget_sam`**, is determined by the current value of the file global variable pointer **`rlm`**.

`X8` is the sole interface to the stream hash function. Its output depends exactly on its current and previous 57 (= (B-P)/8) inputs. `zreg()` means the same as `{int j; for(j=0; j<58; ++j) X8(0);}` —
it merely clears the memory.

`core``(much, who)` slightly improves `malloc` by testing for exhaustion. "`who`" is merely for identifying the failing request.

`GBlc` is a global symbol that counts all blocks of all files. It supports locating stuff in files for later location and perusal.

`IGBl` is a copy of GBlc's value upon opening of the current file.

At the bottom, **`fread`** at the zx label in routine `gaw` in file fl.c reads the files in both phases.

The preprocessor symbol "**`C`**" is for "`const`" and I have tried to put it just about anywhere it could go. This is in line with my suspicion that the default should be with instead of without.

`bgary` is an epiphenomenon, merely for debugging.

`bi` is the index thru the file block buffer, viewed as a word array.

`X8` is the function that does the stationary hash function, one character at a time.

`in` is a file opening variable local to fl.c. It gets its value as routine `fl` is called. It is for the current subject file.

Endianess

This code was designed and debugged on big endian machine (Sparc and 68K). Images of horizontal bit streams inhabit my head as I try to understand the code. I thought it unwise to code it ambidextrously from the beginning.

The main code transformation for a little endian machine is end-for-end swapping of the bits in fields that represent file data. This consists largely of interchanging C's << and >> operators. The two preprocessor symbols `SA` (towards Small Addresses) and `LA` (towards Large Addresses) map to << and >> respectively for big endian machines and vice-versa for little endian ones. When the preprocessor symbol `L` is 1, the high bit of a byte is considered to be adjacent to the low bit of the following byte from the file. It is 1 for little endian machines and 0 for others. If preprocessor symbol `Ebug` is set then “`L` must be wrong for the machine”. This funny mode allows one to check out code pretty well on a machine of the wrong endianess. `Ebug` entails some run-time cost.

The function X8 is an 8 bit interface to the hash magic. It suffices to bit invert both the argument and value of `X8`. We go a bit further for efficiency. The magic constant tables, `head` and `tail`, are special. Their entries are reversed just as they are computed. The internal variables a, b, c, d, y and z, are all reversed. The array index is also reversed. This seemed simpler than reworking the synthetic division logic. `rBy` reverses bytes and `rW` reverses words. The code `X8pr` tests the `X8` code and the code in Fast.c. Its yield depends on L but not the endianess of the machine! It can be debugged on a machine with the wrong end.