Symbolic Alignment Matrix

White paper

Dr. Neal Krawetz

Created for Hewlett-Packard and FOSSology project

7-January-2008

Version 1.1

Copyright

Table of Contents

Revision History

Date	*Version*	*Notes*
10-June-2005	Version 1.0 by Dr. Neal Krawetz for Hewlett-Packard.	Initial draft.
7-January-2008	Version 1.1 by Dr. Neal Krawetz for Hewlett-Packard.	Added notes for bSAM, removed example appendix, updated sections to match current state.

Overview

The Symbolic Alignment Matrix (SAM) compares two sequences of symbols against each other. The system returns whether the two sequences are homologous (similar).

Design Goal

SAM was designed for the task of comparing source code components and identify regions of likely code reuse. For the FOSSology project, code reuse detection is applied toward license phrases: given a set of known license templates, do files contain the known licenses?

For source code reuse, the goal is to identify elements of code that are used in breach of licensing. Examples of potential breaches:

- A function originally released under GPL is reused and released under a non-GPL license. This indicates a breach of the GPL licensing;
- A function originally released as non-GPL is repackaged as GPL. This may violate the non-GPL licensing and/or copyright;
- Proprietary code may be reused and classified as open source; or
- Open source code is reused by proprietary software, breaching the open source licensing.

For license analysis, the goal is to identify which license(s) are associated with which files. Some example problems include:

- Given an unknown file, does it contain any known licenses? If so, which ones and where?
- If the license has a minor modification, what was modified? For example, some developers may take the GPL license and add in the word “Lesser” in an attempt to identify an LGPL license.
- If a license was derived from one or more other licenses, which parts from which known licenses were used?

Purpose

SAM was developed to compare two pieces of source code, or two compiled objects, and determine whether they are likely derivations from the same source. This association is based on two core assumptions:

- **Assumption #1**: The same source code yields the same opcode sequences.
- **Assumption #2**: The same opcode sequences are created by the same source code.

When two developers create the same functionality independently, their source code is likely

unique and generates unique opcode sequences. When code is used as a template, the code generally has a similar appearance but generates different opcode sequences. But when source code is reused (borrowed, leveraged, etc.), there are minor modifications but the primary functional elements remain the same. Since the code is primarily the same, it generates similar opcode sequences. SAM identifies segments of similar code and homologous functionality.

These assumptions may not always be accurate. Issues that may impact alignment include:

- **Macros**. These may enable or disable different source code segments. This is particularly true for debugging statements that are left in source code.
- **Inline code**. Functions denoted as "inline" are usually inserted directly into functions. These may change the opcode sequences, but not necessarily change look of the source code.
- **Compiler optimizations**. Different compiler options may modify opcode sequences for better performance.
- **Code reordering**. If two code segments S_1 and S_2 appear in source code file A as [S_1 S_2] but in a different source code file B as [S_2 S_1], then SAM will only identify the maximum alignment (if $|S_1| > |S_2|$, then SAM will identify $|S_1|$ only.) Other alignment matrices, such as PAM and pam, can identify reordered sequences.

While these issues can (and likely will) result in false-negative matches, in general the assumptions are considered valid:

- Developers generally reuse functions, not individual lines of code.
- Developers may make minor changes to functions, but not significant changes. (Significant changes usually lead to redevelopment rather than reuse.)
- Developers generally do not reorder components in reused source code.
- Developers generally use similar compiler options. For example, nearly all kernel driver developers use the same compiler and compiler flags for the same platform.

This problem space has been extended to the license identification problem. Rather than using source code (or compiled objects), text documents are used. One text document is a known license and the other is the unknown text. If there is a large match between the documents, then it can be assumed that the known license text was used for licensing the unknown text document.

Background

SAM is based on the Protein Alignment Matrix (PAM) by Dayhoff, et al. (1978) and the Program Alignment Matrix (pam) by Krawetz (2002). No source code was referenced or reused in the development of SAM; only concepts and experience were reused.

- Schwartz, R.M. & Dayhoff, M.O. (1978) "Matrices for detecting distant relationships." In "Atlas of Protein Sequence and Structure, vol. 5, suppl. 3," M.O. Dayhoff (ed.), pp. 353-358, Natl. Biomed. Res. Found., Washington, DC.
- Dayhoff, M. O., Schwartz, R. M. & Orcutt, B. C. (1978). "A model of evolutionary change in proteins: matrices for detecting distant relationships." In Atlas of protein sequence and structure, (Dayhoff, M. O., ed.), vol. 5, pp. 345-352. National biomedical research foundation, Washington DC.

PAM consisted of a suite of tools for protein alignment. Programs such as PAM40, PAM120, and PAM250 aligned different sizes of data.

The SAM code uses the same concept behind PAM250 and related systems, with the following differences:

- PAM only permitted 20 different elements (20 amino acids). pam extended PAM to support 256 different binary characters. SAM extends the pam concept to support arbitrary strings (symbols) instead of characters.
- PAM was very inefficient: iterating 3 times through the matrix in order to set values. pam and SAM are efficient: they both iterate once through the matrix. PAM, pam, and SAM all generate the same matrix given the same data.[1]
- PAM used static array sizes and was limited to blocks of 100 amino acid sequences. pam uses a segmented approach with a sliding window, permitting optimal alignment and fast processing for very large data files. SAM uses an allocated matrix, but does not segment data. Thus, very large data files may result in very slow processing and significant memory consumptions.
- PAM used a variable comparison scale, where different amino acid comparisons resulted in different alignment values. (E.g., (ALA,ALA) = 1, but (ALA,ARG)=0.8.) In contrast, both pam and SAM use boolean comparison values to indicate identical or different.
- Both PAM and pam were designed to identify sequences of similarity that may be in reordered sections. SAM strictly looks for code functions – the functions may be in any order, but reordering within a function is not considered similar.

Notations

The two sequences being compared are the *A* and *B* lists. In SAM, both lists contain a sequence of symbols (strings). The length of the lists are denoted by |*A*| and |*B*|, respectively. The alignment matrix is denoted by *A*x*B*. The matrix position (*a,b*) indicates the intersection of *A*[*a*]

[1] Due to different optimization tactics, PAM, pam, and SAM may generate different full matrices, but the optimal paths are identical. In particular, PAM uses no optimizations, pam optimizes for determining the best path in very long sequences, and SAM optimizes for rapidly determining homology – independent of path.

and $B[b]$ within AxB.

SAM determines the percentage of symbols in A that align with symbols in B, and vice versa. These are denoted as $A \rightarrow B$ and $B \rightarrow A$, respectively. For example, if $A \rightarrow B$ is 60%, then it means that 60% of the symbols in A align with symbols in B. Unless A and B are identical and the same length, it is likely that $A \rightarrow B$ and $B \rightarrow A$ will differ.

Homology

PAM was designed to identify homologous sequences. When two sequences are observed with similar elements in a similar order, they are called “homologous.” The term “homology” comes from biochemistry: two proteins are considered homologous when they have similar amino acids in a similar order. The importance being functionality: similar amino acids in a similar order likely have similar properties. PAM was designed to identify regions of similar sequences that indicate likely similar areas of functionality; PAM identifies homologous regions.

The program alignment matrix (pam) applied the same homologous concept to binary files. The system assumes that similar binary sequences yield similar functionality. SAM continues the concept with symbols, such as source code lines or decompiled opcode sequences. While pam does not provide means to identify whether an opcode sequence is data or functional, or to identify platform differences, SAM permits comparing platform independent sequences.

- “Homology” is a boolean concept. There are no "60% homologous" values. Homology is a qualitative description – two sequence either are or are not homologous. “Similarity” is the quantitative description. Two sequences can be 60% similar and be considered homologous. Homology defines the threshold of similarity.
- Homology is asymmetrical. A may be homologous to B when B is not homologous to A. This frequently occurs when $|B|$ is larger than $|A|$.
- Subsets may be homologous. A may not be homologous to B, but the subset a may be homologous to b.

SAM and bSAM

The first instantiation of this system was “SAM”, based on PAM. The current instantiation is “bSAM” or “binary SAM”. SAM was originally created to tokenize documents and compare one file against another. bSAM extends this to the notion of “sections”. A single document may have multiple sections, each requiring a comparison.

SAM was initially developed as a proof-of-concept. bSAM extended the proof to releasable functionality for use with the FOSSology project. As a result, bSAM has many optimizations not found in SAM and a more generic use model. In particular, bSAM only compares sequences of tokens. Filters are used to convert any kind of data file (text, source code, object code, etc.) into tokens for analysis. While SAM identifies the percent of match, bSAM identifies the exact tokens that matched. bSAM also maps tokens to file offsets in the original data file, so the actual match (or missed) sections can be readily identified.

The functional differences between SAM and bSAM are detailed in this document. Since bSAM is an extension to SAM, all algorithm features in SAM also apply to bSAM.

Algorithm

The PAM algorithm is similar to the Unix 'diff' command. The main difference is optimization: 'diff' is significantly optimized and frequently misses corner cases. (Rather than using a matrix, 'diff' maintains state using a short array.) The alignment identified by 'diff' may not necessarily be optimal or symmetrical; 'diff file1 file2' may yield different results than 'diff file2 file1'. In contrast, PAM identifies the optimal alignment and is symmetrical.

The alignment algorithm uses a three-step approach.

Step 1: Initialization

Align symbols to compare in a matrix. For example, let's say the symbols are the letters in *A*:“cheloe” and *B*:“hello ”. The matrix becomes:

	h	***e***	***l***	***l***	***o***
c					
h					
e					
l					
o					
e					

Step 2: Identical identification

Each position within *A*x*B* is marked with “1” if *A*[*a*] is the same as *B*[*b*], and 0 if they are different.

	h	***e***	***l***	***l***	***o***
c					
h	1				
e		1			
l			1	1	
o					1
e		1			

Step 3: Alignment

Each cell in the matrix is updated to be the sum of itself and the maximum value in the sub-matrix [(0,0) , (*a*-1,*b*-1)]. For example:

	h	***e***	***l***	***l***	***o***
c	0	0	0		
h	1	0	0		
e	0	2	1		
l	0	1	3	1	
o					1
e		1			

The table element (2,2), corresponding with the letters 'l' and 'e' was initially zero. It is updated to the value “1” since the maximum value in the subtable [(0,0) , (1,1)] is 1. Similarly, table element (2,3), representing 'l' x 'l' gains the value “3”:

For cell (2,3): identical + max[(0,0) , (1,2)] = 1 + 2 = 3.

The final table becomes:

	h	***e***	***l***	***l***	***o***
c	0	0	0	0	0
h	1	0	0	0	0
e	0	2	1	1	1
l	0	1	3	3	2
o	0	1	2	3	4
e	0	2	2	3	3

The maximum value on the outer edge (4) indicates the maximum number of aligned characters (there are 4 characters aligned).

The cells that contributed to the maximum value indicate the optimal alignment path.

	h	***e***	***l***	***l***	***o***
c	0	0	0	0	0
h	1	0	0	0	0
e	0	2	1	1	1
l	0	1	3	3	2
o	0	1	2	3	4
e	0	2	2	3	3

In this example, the path indicates:

- “4” comes from “o”
- “3” comes from “l” – there are two choices for “hello”, but one choice for “cheloe”.
- “2” comes from “e” (the 1st “e” in "cheloe")
- “1” comes from “h”

The horizontal and vertical gaps show exactly which symbols align:

- “cheloe” aligns with “_helo_”
- “hello” aligns with “hel_o” or “he_lo”

For percentage of alignment:

- cheloe aligns with (4/5 symbols =) 80% of hello.
- hello aligns with (4/6 symbols =) 67% of cheloe.

If |*A*| is much larger than |*B*|, then *B* will likely have a very large percentage, while *A* will have a very lower percentage. If *A* and *B* are both non-trivial (both not small) and both have a high degree of similarity, then they are likely variants of each other.

SAM Symbols

SAM is capable of comparing single letters or symbols. In the case of symbols, each element in the sequence is described by a string. Two elements are considered to be the same only if the strings are identical. The strings can be lines from source code or decompiled assembly code.

SAM Usage

The SAM application has the following usage:

```
Usage: ./sam [options] data1 data2
  Matching options:
    -A percent = percent of data1 that must match (default: -A 90.0)
    -B percent = percent of data2 that must match (default: -B 90.0)
    -C percent = same as '-A percent -B percent'
    -G g = set maximum gap to g (default: -G 5)
    -M m = set minimum sequential to m (default: -M 10)
           -M and -G work together.  We want a sequence of m aligned
           symbols with a gap no larger than g.
  Data processing options:
    -s = use data1 and data2 as strings to compare (default)
    -f = use data1 and data2 as filenames to compare (byte by byte)
    -l = use data1 and data2 as filenames to compare (line by line)
    -m = Data contains multiple data sets, separated by a blank line
         -m only useful with -f or -l.
         Each dataset requires: "a unique label"\n"data".
         A blank line denotes a new data set.
    -p prog = preprocessing program
         -p only useful with -f or -l.
  Debugging options:
    -v = Verbose (-vv = more verbose, etc.)
    -1 = Show matrix stage 1 (same)
    -2 = Show matrix stage 2 (align)
```

SAM displays the degree of similarity. For example, using the command-line:

```
./sam -lm -p filter_C linux-2.6.git/drivers/acorn/block/fd1772.c \
 linux-2.6.git/drivers/block/ataflop.c
```

This command runs SAM on two files: “fd1772.c” and “ataflop.c”. The files are preprocessed using the script “filter_C”. The output from the filter are a set of symbols, one per line (-l), and there are multiple symbol sets (-m) for processing. Each of the symbol tables contains a label (function name) and the symbol set (function data). A blank line separates the end of the sequence from the next. SAM compares every function in “fd1772.c” with ever function in “ataflop.c”. Each function in the first file forms the *A* sets, and each function in the second file forms the *B* sets. A homologous match is determined when:

- $A \rightarrow B$ is at least 90%. (Default command-line “-A 90.0”)
- $B \rightarrow A$ is at least 90%. (Default command-line “-B 90.0”)
- At least 10 symbols in both sets match. (Default command-line “-M 10”) This prevents short sets from creating false-positives in a match.
- The gap between aligned sequences is no more than 5 symbols wide. (Default command-line “-G 5”.) In the previous example, “hello” aligned with “hel_o”, indicating a gap of 1 symbol.

These default settings look for nearly identical matches, indicating source code that only contains

minor changes. The results from the sample command-line indicate one homologous function:

```
***** MATCHED *****
A = linux-2.6.git/drivers/acorn/block/fd1772.c
    finish_fdc_done
B = linux-2.6.git/drivers/block/ataflop.c
    finish_fdc_done
|A| = 94
|B| = 88
max(AxB) = 86
A->B = 91.49%
B->A = 97.73%
```

The results indicate the source of each match (both filename and function since "-m" was specified), the number of symbols in each function, the maximum alignment value, and the percent of alignment.

When compared with the actual source code, the two alignments seem very reasonable (Table 1 and Table 2). The homologous code appears to be reused, with minor changes to function and variable names. The "ataflop.c" function does include a minor code difference: there is a function call for the "else" clause. According to the comments in the source code, "fd1772.c" is based on "ataflop.c".

Table 1. linux-2.6.git/drivers/acorn/block/fd1772.c: finish_fdc_done. Code differences are highlighted.

```
static void finish_fdc_done(int dummy)
{
        unsigned long flags;

        DPRINT(("finish_fdc_done entered\n"));
        STOP_TIMEOUT();
        NeedSeek = 0;

        if (timer_pending(&fd_timer) &&
            time_after(jiffies + 5, fd_timer.expires))
                /* If the check for a disk change is done too early after this
                 * last seek command, the WP bit still reads wrong :-((
                 */
                mod_timer(&fd_timer, jiffies + 5);
        else {
                /*      START_CHECK_CHANGE_TIMER( CHECK_CHANGE_DELAY ); */
        };
        del_timer(&motor_off_timer);
        START_MOTOR_OFF_TIMER(FD_MOTOR_OFF_DELAY);

        save_flags(flags);
        cli();
        /* stdma_release(); - not sure if I should do something DAG  */
        fdc_busy = 0;
        wake_up(&fdc_wait);
        restore_flags(flags);

        DPRINT(("finish_fdc() finished\n"));
}
```

Table 2. linux-2.6.git/drivers/block/ataflop.c: finish_fdc_done. Code changes are highlighted.

```
static void finish_fdc_done( int dummy )
{
        unsigned long flags;

        DPRINT(("finish_fdc_done entered\n"));
        stop_timeout();
        NeedSeek = 0;

        if (timer_pending(&fd_timer) && time_before(fd_timer.expires, jiffies + 5))
                /* If the check for a disk change is done too early after this
                 * last seek command, the WP bit still reads wrong :-((
                 */
                mod_timer(&fd_timer, jiffies + 5);
        else
                start_check_change_timer();
        start_motor_off_timer();

        local_irq_save(flags);
        stdma_release();
        fdc_busy = 0;
        wake_up( &fdc_wait );
        local_irq_restore(flags);

        DPRINT(("finish_fdc() finished\n"));
}
```

bSAM Usage

bSAM has a very similar usage to SAM:

```
Usage: /usr/local/fossology/agents/bsam-engine [options] fileA fileB
  Compares fileA against fileB.
  If either fileA or fileB is -, then a list of files are read from stdin.
  Stdin format: field=value pairs, separated by spaces.
    A=file        :: set fileA to be a pfile ID or regular file.
    B=file        :: set fileB to be a pfile ID or regular file.
    Akey=file_key :: set fileA pfile ID and this is the pfile_pk.
    Bkey=file_key :: set fileB pfile ID and this is the pfile_pk.
    NOTE: stdin can override fileA/fileB set on the command-line!
    NOTE: To use the repository, the corresponding key must be set.
    To turn off the repository, set the key to -1 (default value).
  Matching options:
    -A percent = percent of data1 that must match (default: -A 90)
    -B percent = percent of data2 that must match (default: -B 90)
    -C percent = same as '-A percent -B percent'
    -E = Exhaustive search for best match
    -G g = set maximum gap to g (default: -G 5)
    -L n = set minimum sequence length to check to n (default: -L 10)
    -M m = set minimum sequential to m (default: -M 10)
           -M and -G work together.  We want a sequence of m aligned
           symbols with a gap no larger than g.
    -T t = set a repository type (for -O s and -O t)
    -O f = set output format:
            -O n = Normal DB -- (default -T is 'license')
            -O N = Normal DB -- like '-O n' except uses stdout instead of DB
            -O s = SAM DB -- (default -T is 'sam')
            -O t = Text
  Debugging options:
    -i = Initialize the database, then exit.
    -t file = Test a bsam file (for proper parsing), then exit.
```

The main functional differences between the SAM and bSAM usage:

- bSAM has the option to work with FOSSology's database and scheduler.
- All inputs to bSAM are in the bSAM cache file format. This means, all bSAM filters run independently of the bsam-engine.

bSAM Cache File Format

The bSAM cache file format consists of a series of binary data records. All data records are in the same format:

- Type (2 bytes)
- Size (2 bytes, unsigned) -- size of the data (can be zero meaning "no data")
- Data (size must match Size)

The basic categories for "type" are as follows:

- Type & 0xFFF0 == 0x0000 :: File oriented
- Type & 0xFFF0 == 0x0100 :: Function oriented
- Type & 0xF000 == 0xF000 :: Comment

Specific types are defined as follows:

- 0000 **EOF** -- size and data are not required
- 0001 **File name** -- data is string
- 0002 **File checksum** -- data the checksum (may be a string)
- 0003 **File license** -- data string
- 0004 **File type** -- data string (e.g., "C", "Java", "Class", "Obj"). File type is used to ensure that comparisons are only done between same type of data
- 0010 **File unique value** -- data string
- 0101 **Function name** -- data is string
- 0103 **Function license** -- see 0003 File license
- 0104 **Function type** -- see 0004 File type
- 0108 **Function tokens** -- data contains tokens, 2 bytes each!
- 0110 **Function unique value** -- data string
- 0118 **Function tokens OR list** -- 2 byte tokens, at least one of which must be in the comparison token list. (None are in the comparison? Then skip the comparison!)
- 0128 Function tokens AND list -- 2 byte tokens, all must be in the comparison token list.
- 0129 Function "important" tokens (to be used for choosing between similar matches)
- 0131 Byte offset to start in untokenized file (length is always 4)
- 0132 Byte offset to end in untokenized file (length is always 4)

  Both 0131 and 0132 contain 4 bytes. 0131 and 0132 values should be reset to “undefined” each time 0101 is seen.

- 0138 Byte offsets for tokens. Each token in the 0108 tag will be represented by 1 byte here. This byte is the number of bytes to skip between tokens. This way, matches can be calculated down to specific locations in the file rather than general ranges specified by 0131 - 0132. There is one extra byte here so the end of the last token is known. For example, if the match starts at token #7, then: for(i=0; i<7; i++) RealOffset +=

TokenValue0138[i].

- 0140 Single-sentence licenses (text, tokenized into space separated)
- 01FF End of Function (ok to start processing) -- size is always zero.
- F001 File Comment
- F101 Function Comment
- FFFF General Comment

During file processing, all unknown types are skipped.

For word alignment, add Sizes are padded out to the next 16-bit word boundary. For example, if the Size is 15 then the data contains 15 bytes. Following the data is one “FF” byte to pad out the record to the next word boundary.

Performance

The PAM algorithm is slow. The original PAM implementations iterated through the matrix three times with each sequence being accessed twice. The performance is denoted as $O(|A|^2 \cdot |B|^2)$. SAM has been extremely optimized in order to mitigate the impact from large loops. The matrix is traversed no more than once. This program runs in $O(|A| \cdot |gB|)$, where gB is the maximum gap size for sequence B (see *Optimization: Comparison Truncation*)[2]. The size of gB is no greater than $|B|$ and usually significantly less.

The algorithm optimizations in SAM include:

- Byte pipelining
- Sub-matrix scanning
- Gap truncation
- Comparison restriction
- Comparison truncation
- Comparison omission
- String comparison

The programming convention optimizations include:

- Matrix memory allocation
- Preprocessing vs. caching
- Allocation vs. Memory Mapping

In addition, SAM can be optimized by external factors.

- Calling order

Optimization: Byte Pipelining

The matrix is laid out as one large array: $(a,b) = AxB[a \cdot |B| + b]$. Using this layout, all bytes along the B-axis are sequential. This permits data pipelining due to linear memory accessing.

Optimization: Sub-Matrix Scanning

The original PAM algorithm performed a three-pass approach for filling the matrix. The first pass assigned the 0/1 values for identical data. The second pass updated each cell. Within the

---

[2] Technically, $O(|A| \cdot |gB|)$ is the average case. The worse case occurs when the match and gap thresholds are very large, forcing processing of the entire matrix. This becomes $O(|A|^2 \cdot |B|^2)$. If the gap threshold is relatively small compared to the set size, this becomes $O(|A|^2 \cdot |gB|^2)$ which is still significantly less than $O(|A|^2 \cdot |B|$

second pass, a third pass was used to scan the sub-matrices for the maximum values.

In the optimized code, there is only one pass through the entire matrix. One of the filling properties of the algorithm places the maximum value of any sub-matrix along the outer edge. Thus, scanning the interior of any sub-matrix is unnecessary; only the outer edge needs to be scanned. The algorithms looks like:

```
For each cell (a,b)
      Find the maximum value along the outer edge:
       Edge #1: [(0,b-1) , (a-1,b-1)]
       Edge #2: [(a-1,0) , (a-1,b-1)]
      Add maximum value to identical check: A[a] = B[b]?
      Store the value in the cell.
```

The result is a significant reduction in cells that must be scanned in order to determine the maximum value of any sub-matrix.

Optimization: Gap Truncation

The maximum allowed gap between aligned symbols can be used as a restriction. When determining the value for matrix position (*a*,*b*), *Sub-Matrix Scanning* only checks the outer edges of the sub-matrix: [(*a*-1,0) to (*a*-1,*b*-1)] and [(0,*b*-1) to (*a*-1,*b*-1)]. For large values of *a* or *b*, this scan may take a significant amount of time. As an optimization, we only need to scan the edge that corresponds with the gap. For example, if the maximum allowed gap is 3 (Table 3), then the sub-matrix scan reduces to [(*a*-1,*b*-4) to (*a*-1,*b*-1)] and [(*a*-4,*b*-1) to (*a*-1,*b*-1)].

Table 3. A 7x7 matrix optimized to scan for a maximum gap of 3 symbols.

	S_a	S_b	S_c	S_d	S_e	S_f	S_g
S_a						Gap=5	
S_b						Gap=4	
S_c						Gap=3	
S_d						Gap=2	
S_e						Gap=1	
S_f	Gap=5	Gap=4	Gap=3	Gap=2	Gap=1	Gap=0	
S_g							(*a*,*b*)=?

Optimization: Comparison Restriction

SAM is designed to identify optimal matches, with the “-A” and “-B” options defining the minimum match value. If the minimal match value is large, then only cells along the middle diagonal of the matrix will ever be involved in the match. As such, cells never involved in the match do not need to be scanned. For example, assume we are comparing two sequences of 20

symbols each, and we require a 90% match (Table 2). The match requires that only the maximum value be no less than 16 (90% of 20). Only cells that can lead to a maximum value of 16 are scanned.

Table 4. A 20x20 matrix with a 90% match requirement. Only cells that can lead to the required match are scanned.

	S_a	S_b	S_c	S_d	S_e	S_f	S_g	S_h	S_i	S_j	S_k	S_l	S_m	S_n	S_o	S_p	S_q	S_r	S_s	S_t
S_a	1	1	1	1	1															
S_b	1	2	2	2	2	2														
S_c	1	2	3	3	3	3	3													
S_d	1	2	3	4	4	4	4	4												
S_e	1	2	3	4	5	5	5	5	5											
S_f		2	3	4	5	6	6	6	6	6										
S_g			3	4	5	6	7	7	7	7	7									
S_h				4	5	6	7	8	8	8	8	8								
S_i					5	6	7	8	9	9	9	9	9							
S_j						6	7	8	9	10	10	10	10	10						
S_k							7	8	9	10	11	11	11	11	11					
S_l								8	9	10	11	12	12	12	12	12				
S_m									9	10	11	12	13	13	13	13	13			
S_n										10	11	12	13	14	14	14	14	14		
S_o											11	12	13	14	15	15	15	15	15	
S_p												12	13	14	15	16	16	16	16	16
S_q													13	14	15	16	17	17	17	17
S_r														14	15	16	17	18	18	18
S_s															15	16	17	18	19	19
S_t																16	17	18	19	20

As mentioned above (*Byte Pipelining*), scanning is performed along the *B*-axis. For any position *a*, we only need to scan *b* values between *a*-*mB* and *a*+*mB*, where *mB* is the number of sequences that can be skipped which still achieving the minimum match percentage. In the above example, only four sequences can be skipped (*mB*=4). So we only need to scan from S_{x-4} to S_{x+4} along the *B*-axis, a total of 9 cells per *a* position. This reduced scanning area significantly improves performance, reducing the search space from $|A|\cdot|B|$ to $|A|\cdot mB$.

Optimization: Comparison Truncation

As mentioned in *Byte Pipelining*, the matrix is scanned along the *B*-axis. We can combine this

with the minimum required match for the *A* sequence. In particular, for any column *a*, we can check if the maximum *b* matrix value is sufficient to lead to a match. when a match is not possible, the remainder of the matrix can be skipped.

Using the example from Table 2, the matrix can only achieve the minimal match of 16 if the *A* sequence does not miss more than 4 characters (Table 3). Thus, for row *a*=9, the minimum match value is 9-4=5. We can check if there is some *b* value where the cell value (*a*,*b*) ≥ 5. If the entire set of values [(9,0) to (9,19)] is less than 5, then there is no method for achieving a minimum matrix value of 16.

Table 5. Section of 20x20 table showing the required minimums for a=*9.*

	S_a	S_b	S_c	S_d	S_e	S_f	S_g	S_h	S_i	S_j	S_k	S_l	S_m	S_n	S_o	S_p	S_q	S_r	S_s	S_t
S_f		2	3	4	5	6	6	6	6	6										
S_g			3	4	5	6	7	7	7	7	7									
S_h				4	5	6	7	8	8	8	8	8								
S_i					5	6	7	8	**9**	9	9	9	9							
S_j						6	7	8	9	10	10	10	10	10						
S_k							7	8	9	10	11	11	11	11	11					
S_l								8	9	10	11	12	12	12	12	12				

Assuming the location of the non-matches are random, this optimization would cut the number of comparisons in half. In actuality, the majority of matrices are truncated within the first few rows.

Optimization: Comparison Omission

Similar to *Comparison Restriction*, if the |*A*| and |*B*| are significantly different, it is possible that there can never be a match. For example, if both sequences must match by 90%, but |*A*| < 90%·|*B*|, then there is no possible way these two sequences can match. Thus, the entire matrix filling and comparison is skipped.

Optimization: String Comparisons

The symbols in SAM are stored as strings. Unfortunately, strcmp() is not a fast function. In order to reduce the number of strcmp() calls, each string is prefaced with a one-byte checksum (sum of all of the bytes in the string). For the majority of comparisons, the checksum values will be different, indicating that strcmp() will not match and does not need to be called. The only calls to strcmp() occur when either the strings are identical, or when there is a checksum collision (random odds are $^1/_{256}$).

For the bSAM algorithm, string comparisons are not used at all. Instead, each string is converted to an integer checksum. The comparison becomes a single numerical check (*A*[*a*] = *B*[*b*]?). While

this is a significant speed improvement, there are issues with checksum collisions. However, it is unlikely for a series of checksums to collide identically; only a sequence of checksums in the same order and same context will generate an alignment. Specifically, using a 16-bit integer (and assuming random checksums), there is a "1 in 65,536" chance of a collision. The chance of two random collisions in sequence is 1 in 4,294,967,296. Thus, the likelihood of a whole sequence of collision-matches drops to virtually zero.

Optimization: Matrix Memory Allocations

The single largest memory operation that SAM performs is the matrix. The act of allocating and initializing the matrix can be very time intensive for large sequences. As an optimization, the matrix memory is only allocated when needed.

- **Initialization**. The matrix is not allocated nor initialized unless there is a chance that the matrix may result in a match. As mentioned under *Comparison Omission*, if the sequence sizes differ such that they can never result in a successful alignment, then the matrix does not need to be accessed.
- **Reallocation**. When SAM is used for processing multiple sequences per file (the "-m" parameter for SAM and the default use for bSAM), the matrix is not reallocated unless it needs to be larger. For example, if the first comparison needs 400 cells and the second comparison requires 96 cells, then the matrix memory is not freed or reallocated between sequential calls. The matrix is only reallocated when the comparison requires more than 400 cells.
- **Organization**. The matrix is allocated as a single array, and not as a two-dimensional structure. This permits reuse regardless of the sequence sizes. For example, a 20x20 matrix and 10x40 matrix can both use the same allocated matrix.

In contrast to the matrix allocation, symbols are constantly allocated and freed. Benchmark results indicate that there is no significant optimization for tracking allocation sizes and reallocating for increases during symbol processing. This lack of performance improvement is due to the operating system environment: small memory segments freed by an application are not immediately returned to the global allocation pool. The operating system does not immediately reclaim memory because many applications repeatedly allocate and free small memory sections. The result is a performance gain for both the application and operating system.

Similarly, there is no significant performance gain by memory mapping sequence data files rather than loading them line by line due to file caching.

Optimization: Preprocessing vs. Caching

SAM permits the use of a filtering function ("-p program") for converting the input files into symbol sets. While useful for single executions of SAM, the filters may add significant time to the total execution period. When possible, first run the filters without SAM and store the data (a

filter cache). Then run SAM on the cached data (without the "-p" option).

With the bSAM implementation, the filter program is required and runs independently of the bSAM-engine. First the cache files are created, then they are processed by the bSAM algorithm. In particular, the bSAM-engine never processes the real data file.

Optimization: Allocation vs. Memory Mapping

Reading an input file is a time-consuming operation. When comparing multiple *A* items against multiple *B* items, the outer loop will process each *A* item once, but each *B* item |*A*| times. To reduce the number of reads, bSAM memory maps *A* and *B*. While the looping still occurs, the memory mapped file means (1) the file is not accessed using a sequence if read() calls, and (2) the bSAM token can be used directly from the memory mapped region and not copied to a separate array.

Optimization: Calling Order

As an example, to run SAM and compare all devices in the Linux kernel with all other devices, we could use:

```
for($i=0; $File[$i] ne ""; $i++)
  {
  for($j=$i+1; $File[$j] ne ""; $j++)
    {
    print "Testing $File[$i] $File[$j]\n";
    system "./sam -p ./filter_objdump -lm $File[$i] $File[$j] >> test-all-module
s.out";
    }
  }
```

This function will run the "filter_objdump" filter script twice for every comparison. This filter is written in Perl and has a significant startup code. If there are 1,530 drivers, then SAM will be executed (1,530·1,538) 2,340,900 times. The total runtime on a 1GHz computer will be approximately 2 years. But, by first filtering the data first and then running SAM, the total process time can be reduced to approximately 2 weeks.

A second external optimization relies on symmetry. If the match thresholds are the same for *A* and *B*, then there is no reason to run SAM on both "*A* and *B*" and "*B* and *A*". This symmetry cuts the search space in half. The same 1,530 kernel objects results in 1,170,450 comparisons, or about one week of execution. (With the optimization in bSAM, the time is reduced to a few days.)

Filters

SAM and bSAM only process symbols. The filters convert desired data into a sequence of symbols. There are currently a few types of filters, including Opcode, C, and License.

Filtering Opcode

The opcode filter converts an executable or compiled object into a sequence of symbols with identified functions. This filter operates in multiple stages.

Stage 1: Acquire opcodes

The filter runs the command 'objdump -d' on the executable (or object code). The "-d" parameter decodes the opcode sequences into assembly language. For example, "objdump -d / lib/modules/2.6.10-5-386/kernel/drivers/net/3c509.ko" yields:

```
Disassembly of section .text:

00000000 <el3_common_remove>:
   0:   56                      push   %esi
   1:   53                      push   %ebx
   2:   8b 74 24 0c             mov    0xc(%esp),%esi
   6:   8d 9e 20 02 00 00       lea    0x220(%esi),%ebx
   c:   8b 83 68 01 00 00       mov    0x168(%ebx),%eax
  12:   85 c0                   test   %eax,%eax
  14:   74 07                   je     1d <el3_common_remove+0x1d>
  16:   50                      push   %eax
  17:   e8 fc ff ff ff          call   18 <el3_common_remove+0x18>
  1c:   58                      pop    %eax
  1d:   83 bb 6c 01 00 00 01    cmpl   $0x1,0x16c(%ebx)
  24:   75 0c                   jne    32 <el3_common_remove+0x32>
  26:   ff b3 70 01 00 00       pushl  0x170(%ebx)
  2c:   e8 fc ff ff ff          call   2d <el3_common_remove+0x2d>
  31:   58                      pop    %eax
  32:   56                      push   %esi
  33:   e8 fc ff ff ff          call   34 <el3_common_remove+0x34>
  38:   6a 10                   push   $0x10
  3a:   ff 76 18                pushl  0x18(%esi)
  3d:   68 00 00 00 00          push   $0x0
  42:   e8 fc ff ff ff          call   43 <el3_common_remove+0x43>
  47:   83 c4 10                add    $0x10,%esp
  4a:   89 74 24 0c             mov    %esi,0xc(%esp)
  4e:   5b                      pop    %ebx
  4f:   5e                      pop    %esi
  50:   e9 fc ff ff ff          jmp    51 <el3_common_remove+0x51>
```

The disassembler displays each function name (e.g., "el3_common_remove") and the opcodes within the function, one assembly code element per line.

Stage 2: Identify functions

All non-function and binary code is removed.

```
Function el3_common_remove
. push   %esi
. push   %ebx
. mov    0xc(%esp),%esi
. lea    0x220(%esi),%ebx
. mov    0x168(%ebx),%eax
. test   %eax,%eax
. je     1d <el3_common_remove+0x1d>
. push   %eax
. call   18 <el3_common_remove+0x18>
. pop    %eax
. cmpl   $0x1,0x16c(%ebx)
. jne    32 <el3_common_remove+0x32>
. pushl  0x170(%ebx)
. call   2d <el3_common_remove+0x2d>
. pop    %eax
. push   %esi
. call   34 <el3_common_remove+0x34>
. push   $0x10
. pushl  0x18(%esi)
. push   $0x0
. call   43 <el3_common_remove+0x43>
. add    $0x10,%esp
. mov    %esi,0xc(%esp)
. pop    %ebx
. pop    %esi
. jmp    51 <el3_common_remove+0x51>
```

Each function name is prefaced with the word “Function” and a newline. The newline allows the filter to identify the start of the next function’s code (SAM’s -m option). Each assembly line is prefaced with a period since some functions contain padding (sequences of zeros) that do not resolve to a text opcode. Without the initial period, the padding lines would appear as a newline, incorrectly indicating the start of the next function.

Stage 3: Remove constants

The disassembled code contains link-time constants that may change when the code is reused. For example, the final “jmp 51 <el3_common_remove+0x51>” will likely be different if the code is reused. For simplicity, all constant values are removed.

The result from the filter is a pseudo-unique set of assembly commands. For large sequences (10 or more assembly lines), the likelihood of two different functions having homologous sections without having code reuse drops dramatically. When used with SAM, each line is a symbol (option “-l”) and each filtered output contains multiple functions (SAM option “-m”).

```
Function el3_common_remove
. push %esi
. push %ebx
. mov 0x(%esp),%esi
. lea 0x(%esi),%ebx
. mov 0x(%ebx),%eax
. test %eax,%eax
. je
. push %eax
. call
. pop %eax
. cmpl $0x,0x(%ebx)
. jne
. pushl 0x(%ebx)
. call
. pop %eax
. push %esi
. call
. push $0x
. pushl 0x(%esi)
. push $0x
. call
. add $0x,%esp
. mov %esi,0x(%esp)
. pop %ebx
. pop %esi
. jmp
```

Filtering C

The majority of Linux kernel source code is written in ANSI-C. When people reuse C source code, they usually create some formatting changes:

- White space, such as tabs and indents, are changed.
- Comments are added, modified, or removed.
- Variable and dependent function names may be changed.
- Static values, such as strings or constants may be change.
- Additional lines may be added, or existing lines may be removed.

In contrast, some factors are rarely changed:

- The number and relative position of tokens (symbols, parenthesis, scopes, mathematical operators) remain homologous.
- The number and relative position of variables, strings, and functions remain the homologous. For example, code reuse may change a function's name, but a function will rarely be changed into a variable or string.
- The number and type of function parameters will remain homologous.

The C filter converts source code into a set of functions and pseudo-unique tokens that can be used with SAM's "-l" and "-m" options. The filter operates in multiple stages.

Stage 1: Remove comments and tag scopes

The entire source file is loaded into memory. As it is loaded, the data is preprocessed.

- All ANSI-C comments are removed. ("/* ... */")
- All C++ comments are removed ("// ...")
- All compiler directives are removed. These are denoted by a "#" at the beginning of the line. Some examples include "#include", "#define", and "#if". (Spaces may preface or follow the "#", but the "#" will be the first active element on the line.)
- Newlines are removed. All occurrences of line concatenation ("\" at the end of a line) are placed with spaces.
- Every begin and end quote and double-quote is identified. Double quotes "..." are replaced with "< ... >" to indicate start and stop. Similarly, single quotes '...' are replaced with '< ... >'. The replacement indicators are prefaced and appended with spaces to ensure no accidental matches.
- Every new scope is marked with the current scope depth. For example "{ ([ ]) }" becomes "{0 (1 [2]2)1 }0". This is required for identifying functions.

The result of this preprocessing is a single line containing all of the active source code objects. The line includes tags to identify matching scopes and strings.

Stage 2: Identify functions

Function appear as a word space followed by a set of 0-scope parenthesis "(0 ...)0" and a set of 0-scope braces "{0 ... }0". All non-functional elements are removed. The items removed include:

- Function prototypes: int hello(int i);
- Structure definitions: struct hello { int i; };
- Global variables
- Macros
- Function return parameter definitions
- Any variable definitions between the parenthesis and braces are removed. (C&R style parameter definitions.)

The result of this stage is a long string containing a function name and the function scope.

The next two stages operate within each identified function scope, defined by the range "{0 ... } 0".

Stage 2.1: Separate tokens

The C preparser permits some tokens to be adjacent to variable or function names. To simplify the filtering process, tokens are distinctly separated. Examples include:

- Semi-colons: spaces are added before and after every semi-colon
- Commas
- ++ and -- assignment operators are padded with whitespace
- All word tokens (one or more sets of [A-Za-z0-9_]) are padded with whitespace.

Unfortunately, some items get separated that should not be. So a second phase puts back some parameters. These include:

- Any scope followed by a number has the space removed. ("{ 0" becomes "{0".
- Complex variables are joined. For example, "A -> B" becomes "A->B" and "A . B" becomes "A.B". This ensures that complex variables are still treated as a single variable. When code is reused, variables may be replaced with different variable or complex variables.

Stage 2.2: Replace variables

All variable and function names are simply replaced with the word "Var". For the homology comparison, the reuse of variables is unimportant – only the frequency and position of variables.

Although this replacement makes no distinction between variables and function, functions are still identifiable by the parenthesis after the "Var" string.

The only variable not replaced are strings that are specific to the C language. Those few strings denote specific functional operators. The complete set of operator strings are[3]:

```
auto double int struct break else long switch case enum register typedef char extern
return union const float short unsigned continue for signed void default goto sizeof
volatile do if static while
```

Stage 2.3: Replace strings

All strings, denoted by "< ... >" and '< ... >', are replaced. Double-quoted strings are placed with the text "String", and single-quotes are placed with "Char". Similarly, numbers (denoted by [0-9] + or 0x[0-9]+) are replaced with "Num".

Some programmers take shortcuts, using adjacent strings or numbers. For example, 'printf ("hello" "there")'. All instances of "String String" or "Num Num" are reduced to a single String

[3] Source: http://publications.gbdirect.co.uk/c_book/chapter2/keywords_and_identifiers.html

or Num, respectively.

Stage 2.4: Remove variable definitions

There are four types of actions within a function:

1. Assignments: a=b, a++
2. Operators: a>b, a+b
3. Functions: a(b)
4. Declarations: int a;

Declarations are the only lines within a function that generate no opcode sequences. Technically, declarations assign data space, not code space, within a compiled object. These account for static offsets within the assembled code. Since they generate no opcode sequences, they are removed.

Stage 3: Replace parameters

Within the function definition may be parameters "(0 ...)0". During code reuse, the parameter types ("int" or "struct complex *") and variable names may vary, but the number of parameters remains homologous. A few parameters may be added or removed, but the overall number of parameters remains the same.

Each parameter – both prototype and name – is replaced with the word "Parm". The exception are void parameters which are removed. For example:

- "myfun (0 int a , char * b)0" becomes "myfun (0 parm , parm)0".
- " myfun (0 void 0)" becomes "myfun (0 0)".

Stage 4: Remove tags

Reused source code may include additional scopes. To prevent scope tags from being used all scope depth markers are removed. For example, "{0 ... }0" becomes "{ ... }".

Net result

The final result of the filtering is a file containing function names and tokens, with one token per line. (Appropriate for use with SAM's "-m" and "-l" options).

For example, here is a section of source code from "sam.c".

Table 6. Sample code from "sam.c".

```
#define Max(a,b)        ((a) > (b) ? (a) : (b))

/*******************************************
 PrintMatrix(): display the matrix for debugging.
 *******************************************/
void    PrintMatrix     ()
{
  int a,b,aoffset;

  /* display header across */
  printf("   ");
  for(b=0; b<MaxSymbol[1]; b++)
    {
    printf(" %3c",Symbol[1][b][1]);
    }
  printf("\n");

  for(a=0; a<MaxSymbol[0]; a++)
    {
    printf(" %3c",Symbol[0][a][1]); /* header */
    aoffset = a*MaxSymbol[1];
    for(b=0; b<MaxSymbol[1]; b++)
      {
      printf("%3d ",Matrix[aoffset+b]);
      }
    printf("\n");
    }
} /* PrintMatrix() */
```

After filtering, the code only contains symbols indicating C tokens, variables, and strings.

Table 7. Sample code from "sam.c" after being processed by the filter.

```
PrintMatrix
(
)
{
Var
(
String
)
;
for
(
Var
=
Num
;
Var
<
Var
[
Num
]
;
Var
++
)
{
Var
(
String
,
Var
[
Num
]
[
Var
]
[
Num

...
```

The sequence of symbols per function are pseudo-unique per code functionality. For large sequences, the chances of two vastly different functions have similar token sequences becomes very low.

Filtering Licenses

The license filter is used by the FOSSology project for comparing text strings and identifying licenses. The basic filter tokenizes the file based on sequence of characters. For example, a sequence of letters is a token, punctuation form a token, and numbers form a token. Thus, “Built-in code 2005” forms five tokens: “built”, “-”, “in”, “code”, “2005”.

For consistency, some strings are modified. For example, all text is converted to lowercase. In places where text variations are permitted, tokens are normalized. For example, “copyright” and “(C)” are equivalent and converted to the same string. Also, any 4-digit number is converted to a generic year. This allows alignment of phrases such as “Copyright 2005 Hewlett-Packard”.

For bSAM, each tokenized string is converted to a simple 16-bit checksum. The checksum is generated using a quick-and-dirty summation of the characters in the string, and not using a standard checksum algorithm such as CRC-16. In particular, pairs of letters in the string are treated as 16-bit integers and added together: “hello” becomes “he” + “ll” + “o[null]” (as 16-bit values yields 0x6865 + 0x6C6C + 0x6F00 = 0x43D1). While a variation of the letter positions, such as “llheo” or “hflko”, will generate the same simple checksum, this is unlikely to occur within normal writing conventions. This simple checksum is significantly faster than CRC-16, and it is effectively as accurate. However, should the simple checksum pose an issue, it can easily be replaced with a different checksum algorithm. (bSAM only compares sequences of 16-bit integers and does not care how the integers are generated.)

The result from the Filter_License code are a sequence of tokens for use with the alignment matrix. A match indicates the same words appearing in the same context. License identification occurs when *A* is an unknown text sample and *B* is a series of known licenses. Changing a single word in a license (e.g., replacing the string “GPL” with “Neal”) will not significantly alter the match. Moreover, optimal alignment path identifies exactly which tokens matched, allowing the identification of any text changes.

Extending Filters

The SAM and bSAM implementations only require input tokens. Since filters are used to convert any kind of input into a sequence of tokens, anything can be compared with anything.

For example, with a little modification, the C filter became a Java filter that processes Java code. Similarly, the Java class filter is a variation of the objdump filter; it uses ‘javac’ to extract opcode sequences and creates tokens for comparing compiled Java.

Filters for other programming languages, including as Lisp, Pascal, and JavaScript, can be readily constructed. For the SAM (and bSAM) comparison, code of the same type should be compared (e.g., comparing C to C, or Java to Java), while dissimilar languages should not be compared (e.g., Java to C).

Appendix A: Example Opcode Matches

When comparing object code, very small functions may appear similar. Examples include simply functions that initialize one or two variables with static values. A minimum match size of "-M 10" reduces the chances of matching very small functions. This does not remove all false positives, but drastically reduces the quantity.

Within the Linux 2.6 drivers are a significant amount of code reuse. The reuse appears attributed to three factors:

- **Same developer**. Within the kernel code, a small set of developers created a large number of drivers.
- **Template code**. Public templates for initialization, exit, read, and write appear as similar code.
- **Compilation inclusion**. Many drivers include the same source files during compilation. This usually appears as homologous functions with the same function name.
- **Compilation macros**. Driver macros, such as lock functions, appear homologous between drivers.

Example: AGP

An example device driver match comes from the "intel-mch-agp.ko" and "nvidia-agp.ko" drivers.

```
***** MATCHED *****
A = /lib/modules/2.6.10-5-386/kernel/drivers/char/agp/intel-mch-agp.ko
    Function intel_8xx_fetch_size
B = /lib/modules/2.6.10-5-386/kernel/drivers/char/agp/nvidia-agp.ko
    Function nvidia_fetch_size
|A| = 35
|B| = 36
max(AxB) = 34
A->B = 97.14%
B->A = 94.44%
```

These homologous drivers align with 34 assembly code lines. The source files (Table 8 and Table 9) appear similar, and both are GPL licensed. The major differences between these functions are the calling parameters.

Table 8. linux-2.6.git/drivers/char/agp/intel-agp.c. Differences with nvidia-agp.c are highlighted.

```
static int __intel_8xx_fetch_size(u8 temp)
{
        int i;
        struct aper_size_info_8 *values;

        values = A_SIZE_8(agp_bridge->driver->aperture_sizes);

        for (i = 0; i < agp_bridge->driver->num_aperture_sizes; i++) {
                if (temp == values[i].size_value) {
                        agp_bridge->previous_size =
                                agp_bridge->current_size = (void *) (values
+ i);
                        agp_bridge->aperture_size_idx = i;
                        return values[i].size;
                }
        }
        return 0;
}
```

Table 9. linux-2.6.git/drivers/char/agp/nvidia-agp.c. Differences with intel-agp.c are highlighted.

```
static int nvidia_fetch_size(void)
{
        int i;
        u8 size_value;
        struct aper_size_info_8 *values;

        pci_read_config_byte(agp_bridge->dev, NVIDIA_0_APSIZE,
&size_value);
        size_value &= 0x0f;
        values = A_SIZE_8(agp_bridge->driver->aperture_sizes);

        for (i = 0; i < agp_bridge->driver->num_aperture_sizes; i++) {
                if (size_value == values[i].size_value) {
                        agp_bridge->previous_size =
                                agp_bridge->current_size = (void *) (values
+ i);
                        agp_bridge->aperture_size_idx = i;
                        return values[i].size;
                }
        }

        return 0;
}
```

Appendix B: Example C Matches

When comparing source files, small functions frequently appear similar. A minimum match size of "-M 35" reduces the chances of matching small functions. This does not remove all false positives, but drastically reduces the quantity.

Within the Linux 2.6 drivers are a significant amount of code reuse. Most reuse is attributed to the same developer creating multiple drivers. In other cases, the developer references the source file. But there are a few examples of reuse without attribution. An example C source code match comes from the ACPI "scan.c" and hotplug "pci_hotplug_core.c" source files.

```
***** MATCHED *****
A = linux-2.6.git/drivers/acpi/scan.c
    acpi_device_attr_show
B = linux-2.6.git/drivers/pci/hotplug/pci_hotplug_core.c
    acpi_device_attr_show
|A| = 41
|B| = 41
max(AxB) = 41
A->B = 100.00%
B->A = 100.00%

***** MATCHED *****
A = linux-2.6.git/drivers/acpi/scan.c
    acpi_device_attr_store
B = linux-2.6.git/drivers/pci/hotplug/pci_hotplug_core.c
    hotplug_slot_attr_store
|A| = 45
|B| = 45
max(AxB) = 44
A->B = 97.78%
B->A = 97.78%
```

According to the alignment matrix, acpi_device_attr_show() is the same as acpi_device_attr_show(), and acpi_device_attr_store() is homologous (but not identical) to hotplug_slot_attr_store(). The source code for these functions are listed in Table 4 and Table 5.

Although the two files use different function names, all other attributes including spacing, variable names, and comments (or lack of) are identical – even though SAM only tracked tokens. The only significant different is the replacement of the final "len" to "0" (filtered to "Var" and "Num").

More interesting are the copyright credits for these source files. "scan.c" contains no copyright, no credit, and no licensing information. In contrast, "pci_hotplug_core.c" is GPL licensed with copyright credit assigned to IBM. Judging strictly from the source code, it is unclear which came first or whether they were both based on another common source.

Table 10. linux-2.6.git/drivers/acpi/scan.c. Differences with pci_hotplug_core.c are highlighted.

```
static ssize_t acpi_device_attr_show(struct kobject *kobj,
                struct attribute *attr, char *buf)
{
        struct acpi_device *device = to_acpi_device(kobj);
        struct acpi_device_attribute *attribute = to_handle_attr(attr);
        return attribute->show ? attribute->show(device, buf) : 0;
}
static ssize_t acpi_device_attr_store(struct kobject *kobj,
                struct attribute *attr, const char *buf, size_t len)
{
        struct acpi_device *device = to_acpi_device(kobj);
        struct acpi_device_attribute *attribute = to_handle_attr(attr);
        return attribute->store ? attribute->store(device, buf, len) : len;
}
```

Table 11. linux-2.6.git/drivers/pci/hotplug/pci_hotplug_core.c. Differences with scan.c are highlighted.

```
static ssize_t hotplug_slot_attr_show(struct kobject *kobj,
                struct attribute *attr, char *buf)
{
        struct hotplug_slot *slot = to_hotplug_slot(kobj);
        struct hotplug_slot_attribute *attribute = to_hotplug_attr(attr);
        return attribute->show ? attribute->show(slot, buf) : 0;
}

static ssize_t hotplug_slot_attr_store(struct kobject *kobj,
                struct attribute *attr, const char *buf, size_t len)
{
        struct hotplug_slot *slot = to_hotplug_slot(kobj);
        struct hotplug_slot_attribute *attribute = to_hotplug_attr(attr);
        return attribute->store ? attribute->store(slot, buf, len) : 0;
}
```

The changes between the two files can be achieved by:

- Replacing all instances of the string “device” with “slot”.
- Replacing all instances of the string “acpi” with “hotplug”.
- Replacing all instances of the string “handle” to “hotplug”.

Although these functions could be coincidentally similar – they are relatively small and perform similar functions – the similarities are more likely due to code reuse.