Evaluation

Rybka's evaluation has been the subject of much speculation ever since its appearance. Various theories have been put forth about the inner workings of the evaluation, but with the publication of Strelka, it was shown just how wrong everyone was. It is perhaps ironic that Rybka's evaluation is its most similar part to Fruit; it contains, in my opinion, the most damning evidence of all.

General Differences

Simply put, Rybka's evaluation is virtually identical to Fruit's. There are a few important changes though, that should be kept in mind when viewing this analysis.

- Most obviously, the translation to Rybka's bitboard data structures. In some instances, such as in the pawn evaluation, the bitboard version will behave slightly differently than the original. But the high-level functionality is always equivalent in these cases; the changes are brought about because of a more natural representation in bitboards, or for a slight speed gain. In other cases the code has been reorganized a bit; this should be seen more as an optimization than as a real change, since the end result is the same.
- All of the endgame and draw recognition logic in Fruit has been replaced by a large material table in Rybka. This serves mostly the same purpose as the material hash table in Fruit, since it has an evaluation and a flags field.
- All of the weights have been tuned. Due to the unnatural values of Rybka's evaluation parameters, they were mostly likely tuned in some automated fashion. However, there are a few places where the origin of the values in Fruit is still apparent: piece square tables, passed pawn scores, and the flags in the material table.

Evaluation Detail

In the following pages I will go into more depth about the details of each aspect of the evaluations and their similarities and differences.

- Pawn evaluation: `pawn_get_info()`
- Piece evaluation: `eval_piece()`
- King Safety/Shelter: `eval_king()`
- Passed Pawns: `eval_passer()`
- Patterns: `eval_pattern()`
- Material

Piece Square Tables

Piece square tables are a very simple technique used for basic evaluation. For every piece type and square, PSTs have a value for that piece being on that square. Fruit uses a clear and simple but effective way of calculating the tables. Looking at Rybka's PSTs, we will see that they are calculated using these exact same constants except with different weights. Also, note that here too that the PST values are hardcoded into the Rybka executable file, they are not calculated at startup like Fruit's. The code shown here is simply the functional equivalent; it calculates the Rybka PSTs.

Constants

Fruit's PSTs are based on a small set of constants, which allow for a compact representation of the values. For most pieces, the entire set of 64 squares is compressed into 16 constants (8 for ranks, 8 for files) plus two weights.

Constants in Fruit

```
static const int PawnFile[8] = { -3, -1, +0, +1, +1, +0, -1, -3 };
static const int KnightLine[8] = { -4, -2, +0, +1, +1, +0, -2, -4 };
static const int KnightRank[8] = { -2, -1, +0, +1, +2, +3, +2, +1 };
static const int BishopLine[8] = { -3, -1, +0, +1, +1, +0, -1, -3 };
static const int RookFile[8] = { -2, -1, +0, +1, +1, +0, -1, -2 };
static const int QueenLine[8] = { -3, -1, +0, +1, +1, +0, -1, -3 };
static const int KingLine[8] = { -3, -1, +0, +1, +1, +0, -1, -3 };
static const int KingFile[8] = { +3, +4, +2, +0, +0, +2, +4, +3 };
static const int KingRank[8] = { +1, +0, -2, -3, -4, -5, -6, -7 };
```

Pawns

First we have pawns. The pawn PSTs are just based on the file. We also add in a bonus for some of the center squares. Rybka is the same, but it adds in an endgame bonus, and also only the bonuses for D5/E5 are added.

Fruit

```
static const int PawnFileOpening = 5;

...

for (sq = 0; sq < 64; sq++) {
 P(piece,sq,Opening) +=
PawnFile[square_file(sq)] *
 PawnFileOpening;
}

P(piece,D3,Opening) += 10;
P(piece,E3,Opening) += 10;

P(piece,D4,Opening) += 20;
P(piece,E4,Opening) += 20;

P(piece,D5,Opening) += 10;
P(piece,E5,Opening) += 10;
```

Rybka

```
static const int PawnFileOpening = 181;
static const int PawnFileEndgame = -97;

...

for (sq = 0; sq < 64; sq++) {
 P(piece,sq,Opening) +=
PawnFile[square_file(sq)] *
 PawnFileOpening;
 P(piece,sq,Endgame) +=
PawnFile[square_file(sq)] *
 PawnFileEndgame;
}

P(piece,D5,Opening) += 74;
P(piece,E5,Opening) += 74;
```

Knights

Next there are knights. Knight PSTs are based on the rank and file, with a "center" term counting for both ranks and files, and also a separate rank bonus. Two corrections are then applied: a "trapped" penalty for knights on A8/H8, and a "back rank" penalty for knights on the first rank (to help development). Also note that the "back rank" penalty has a weight of 0 in both programs, so it doesn't appear in the PSTs.

Fruit

```
static const int KnightCentreOpening = 5;
static const int KnightCentreEndgame = 5;
static const int KnightRankOpening = 5;
static const int KnightBackRankOpening = 0;
static const int KnightTrapped = 100;

...

for (sq = 0; sq < 64; sq++) {
 P(piece,sq,Opening) +=
KnightLine[square_file(sq)] *
 KnightCentreOpening;
 P(piece,sq,Opening) +=
KnightLine[square_rank(sq)] *
 KnightCentreOpening;
 P(piece,sq,Endgame) +=
KnightLine[square_file(sq)] *
 KnightCentreEndgame;
 P(piece,sq,Endgame) +=
KnightLine[square_rank(sq)] *
 KnightCentreEndgame;
}
for (sq = 0; sq < 64; sq++) {
 P(piece,sq,Opening) +=
KnightRank[square_rank(sq)] *
 KnightRankOpening;
}
for (sq = A1; sq <= H1; sq++) {
 P(piece,sq,Opening) -= KnightBackRankOpening;
}
P(piece,A8,Opening) -= KnightTrapped;
P(piece,H8,Opening) -= KnightTrapped;
```

Rybka

```
static const int KnightCentreOpening = 347;
static const int KnightCentreEndgame = 56;
static const int KnightRankOpening = 358;
static const int KnightBackRankOpening = 0;
static const int KnightTrapped = 3200;

...

for (sq = 0; sq < 64; sq++) {
 P(piece,sq,Opening) +=
KnightLine[square_file(sq)] *
 KnightCentreOpening;
 P(piece,sq,Opening) +=
KnightLine[square_rank(sq)] *
 KnightCentreOpening;
 P(piece,sq,Endgame) +=
KnightLine[square_file(sq)] *
 KnightCentreEndgame;
 P(piece,sq,Endgame) +=
KnightLine[square_rank(sq)] *
 KnightCentreEndgame;
}
for (sq = 0; sq < 64; sq++) {
 P(piece,sq,Opening) +=
KnightRank[square_rank(sq)] *
 KnightRankOpening;
}
for (sq = A1; sq <= H1; sq++) {
 P(piece,sq,Opening) -= KnightBackRankOpening;
}
P(piece,A8,Opening) -= KnightTrapped;
P(piece,H8,Opening) -= KnightTrapped;
```

Bishops

Next are the bishops. Bishop PSTs are based on the rank and file, with a "center" term counting equally for both ranks and files. There is also a bonus for being on either of the main diagonals, and there is an additional penalty for being on the back rank.

Fruit

```
static const int BishopCentreOpening = 2;
static const int BishopCentreEndgame = 3;
static const int BishopBackRankOpening = 10;
static const int BishopDiagonalOpening = 4;
...

for (sq = 0; sq < 64; sq++) {
 P(piece,sq,Opening) +=
BishopLine[square_file(sq)] *
 BishopCentreOpening;
 P(piece,sq,Opening) +=
BishopLine[square_rank(sq)] *
 BishopCentreOpening;
 P(piece,sq,Endgame) +=
BishopLine[square_file(sq)] *
 BishopCentreEndgame;
 P(piece,sq,Endgame) +=
BishopLine[square_rank(sq)] *
 BishopCentreEndgame;
}
for (sq = A1; sq <= H1; sq++) {
 P(piece,sq,Opening) -= BishopBackRankOpening;
}
for (i = 0; i < 8; i++) {
 sq = square_make(i,i);
```

Rybka

```
static const int BishopCentreOpening = 147;
static const int BishopCentreEndgame = 49;
static const int BishopBackRankOpening = 251;
static const int BishopDiagonalOpening = 378;

...

for (sq = 0; sq < 64; sq++) {
 P(piece,sq,Opening) +=
BishopLine[square_file(sq)] *
 BishopCentreOpening;
 P(piece,sq,Opening) +=
BishopLine[square_rank(sq)] *
 BishopCentreOpening;
 P(piece,sq,Endgame) +=
BishopLine[square_file(sq)] *
 BishopCentreEndgame;
 P(piece,sq,Endgame) +=
BishopLine[square_rank(sq)] *
 BishopCentreEndgame;
}
for (sq = A1; sq <= H1; sq++) {
 P(piece,sq,Opening) -= BishopBackRankOpening;
}
for (i = 0; i < 8; i++) {
 sq = square_make(i,i);
```

` P(piece,sq,Opening) += BishopDiagonalOpening;`	` P(piece,sq,Opening) += BishopDiagonalOpening;`
` P(piece,square_opp(sq),Opening) +=`	` P(piece,square_opp(sq),Opening) +=`
`BishopDiagonalOpening;`	`BishopDiagonalOpening;`
`}`	`}`

Rooks

Next are the rooks. Rooks PSTs are very simple, and are only based on the file.

Fruit	Rybka
`static const int RookFileOpening = 3;`	`static const int RookFileOpening = 104;`
`...`	`...`
`for (sq = 0; sq < 64; sq++) {`	`for (sq = 0; sq < 64; sq++) {`
` P(piece,sq,Opening) +=`	` P(piece,sq,Opening) +=`
`RookFile[square_file(sq)] *`	`RookFile[square_file(sq)] *`
` RookFileOpening;`	` RookFileOpening;`
`}`	`}`

Queens

Next are the queens. Queens are based on the center bonus (weighting rank and file equally), with an additional correction for being on the back rank.

Fruit	Rybka
`static const int QueenCentreOpening = 0;`	`static const int QueenCentreOpening = 98;`
`static const int QueenCentreEndgame = 4;`	`static const int QueenCentreEndgame = 108;`
`static const int QueenBackRankOpening = 5;`	`static const int QueenBackRankOpening = 201;`
`...`	`...`
`for (sq = 0; sq < 64; sq++) {`	`for (sq = 0; sq < 64; sq++) {`
` P(piece,sq,Opening) +=`	` P(piece,sq,Opening) +=`
`QueenLine[square_file(sq)] *`	`QueenLine[square_file(sq)] *`
` QueenCentreOpening;`	` QueenCentreOpening;`
` P(piece,sq,Opening) +=`	` P(piece,sq,Opening) +=`
`QueenLine[square_rank(sq)] *`	`QueenLine[square_rank(sq)] *`
` QueenCentreOpening;`	` QueenCentreOpening;`
` P(piece,sq,Endgame) +=`	` P(piece,sq,Endgame) +=`
`QueenLine[square_file(sq)] *`	`QueenLine[square_file(sq)] *`
` QueenCentreEndgame;`	` QueenCentreEndgame;`
` P(piece,sq,Endgame) +=`	` P(piece,sq,Endgame) +=`
`QueenLine[square_rank(sq)] *`	`QueenLine[square_rank(sq)] *`
` QueenCentreEndgame;`	` QueenCentreEndgame;`
`}`	`}`
`for (sq = A1; sq <= H1; sq++) {`	`for (sq = A1; sq <= H1; sq++) {`
` P(piece,sq,Opening) -= QueenBackRankOpening;`	` P(piece,sq,Opening) -= QueenBackRankOpening;`
`}`	`}`

Kings

Lastly, we evaluate the king. In the opening, we have bonuses for the rank and file, and in the endgame, there is simply a center bonus.

Fruit	Rybka
`static const int KingCentreEndgame = 12;`	`static const int KingCentreEndgame = 401;`

```
static const int KingFileOpening = 10;
static const int KingRankOpening = 10;

...

for (sq = 0; sq < 64; sq++) {
 P(piece,sq,Endgame) +=
KingLine[square_file(sq)] *
 KingCentreEndgame;
 P(piece,sq,Endgame) +=
KingLine[square_rank(sq)] *
 KingCentreEndgame;
}
for (sq = 0; sq < 64; sq++) {
 P(piece,sq,Opening) +=
KingFile[square_file(sq)] *
 KingFileOpening;
}
for (sq = 0; sq < 64; sq++) {
 P(piece,sq,Opening) +=
KingRank[square_rank(sq)] *
 KingRankOpening;
}
```

```
static const int KingFileOpening = 469;
static const int KingRankOpening = 0;

...

for (sq = 0; sq < 64; sq++) {
 P(piece,sq,Endgame) +=
KingLine[square_file(sq)] *
 KingCentreEndgame;
 P(piece,sq,Endgame) +=
KingLine[square_rank(sq)] *
 KingCentreEndgame;
}
for (sq = 0; sq < 64; sq++) {
 P(piece,sq,Opening) +=
KingFile[square_file(sq)] *
 KingFileOpening;
}
for (sq = 0; sq < 64; sq++) {
 P(piece,sq,Opening) +=
KingRank[square_rank(sq)] *
 KingRankOpening;
}
```

Conclusion

We have found that, looking at the PST values of Fruit and Rybka, that Rybka's PSTs can be calculated using Fruit's code with a minimum of changes. The only differences are the various weights (the constants found near the top of pst.cpp in Fruit) and the bonuses for center pawns. Because of Fruit's unique PST initialization code, the origin of Rybka's PSTs in Fruit is clear.

Passed Pawn Evaluation

Rybka's passed pawn evaluation was originally thought to be extremely complex. In reality though, it's really very simple. Here I will show that the passed pawn evaluation is equivalent to Fruit's, except for different weights, using a quick approximation of the static exchange evaluation, and the division of the free pawn bonus into three separate bonuses.

Quad

In showing the equivalence between Fruit and Rybka's passed pawn evaluation, it is first necessary to understand the `quad()` function in Fruit. `quad()` calculates a bonus for a passed pawn based on a minimum and a maximum, with the final score being based on the rank of the pawn. It does this using a lookup table with a value from 0 to 256. This is used as a ratio (over 256) which is how far between the minimum and maximum the final score is. If the pawn is on the 7th rank, it gets the full bonus; if it is on the 2nd or 3rd, it gets the minimum. On ranks 4-6 it gets somewhere in between.

`quad()` in Fruit

```
for (rank = 0; rank < RankNb; rank++) Bonus[rank] = 0;

Bonus[Rank4] = 26;
Bonus[Rank5] = 77;
Bonus[Rank6] = 154;
Bonus[Rank7] = 256;

...

int quad(int y_min, int y_max, int x) {
 int y;
 y = y_min + ((y_max - y_min) * Bonus[x] + 128) / 256;
 return y;
}
```

In Fruit, there are several bonuses for a passed pawn. The endgame score has a fixed minimum, and all the bonuses for the pawn simply increase the maximum. This is equivalent to adding a set endgame bonus with `quad()` (between `PassedEndgameMin` and `PassedEndgameMax`) and using `quad()` for each subsequent bonus with a minimum of 0 and a max of the bonus. This has been implemented in an optimized (and slightly confusing) way in Fruit. To illustrate this, here is Fruit's evaluation and a simplified version. They both produce the same output, but the simplified version is a bit slower.

Endgame passed pawn evaluation in Fruit

```
min = PassedEndgameMin;
max = PassedEndgameMax;
delta = max - min;

...

// misc. bonuses
delta += bonus;

...

eg[att] += min;
if (delta > 0) eg[att] += quad(0,delta,rank);
```

Simplified version

```
eg[att] +=
quad(PassedEndgameMin,PassedEndgameMax,rank);

...

// misc. bonuses
eg[att] += quad(0,bonus,rank);
```

Opening

The opening value for passed pawns is very simple in both programs: we simply add a fixed bonus based on the rank.

Fruit	Rybka
`op[att] += quad(PassedOpeningMin,PassedOpeningMax,rank);`	`opening += PassedOpening[rank];`

Endgame

Rybka and Fruit both have the same basic structure in the endgame passed pawn scoring: calculate a minimum and a maximum value and interpolate the real value based on the rank of the pawn. See above for details regarding Fruit; Rybka works the same way. We start out initializing the minimum:

Fruit	Rybka

`min = PassedEndgameMin;`
`...`
`eg[att] += min;`
	`endgame += PassedEndgame[rank];`

Dangerous Bonuses

Next, we add the "dangerous" bonuses to the endgame maximum. There are actually a few of these: if the opponent has no pieces, we detect whether the passer is unstoppable, or if our king is in a position to protect it while promoting. We then check if the passer is free; that is, it can walk to the promotion square without being blocked or captured.

The unstoppable passer is simply a passer that isn't blocked by a friendly piece and the opponent king is outside its "square". The king passer is a passer on the 6th or 7th rank which the king defends while simultaneously defending the promotion square. These defintions are exact bitboard equivalents in Rybka, and they both receive the same bonus of `UnstoppablePasser`. This bonus is not based on rank like the other bonuses, but is simply the value of a queen minus the value of a pawn.

Next is the free pawn. The opponent has pieces in this case to potentially keep our pawn from promoting, so we need to check if it can escape. In Fruit, the square in front of the pawn must be empty, and the pawn must be able to advance safely there. We use the static exchange evaluation (SEE) to make sure that even if the square is attacked by the opponent, we can recapture on the square. This check is only done on the square directly in front of the pawn in Fruit, but since Rybka is bitboard based, we can quickly do the same calculation for all squares in front of the pawn up to the promotion square. To do this we make an approximation of the SEE that is usually equivalent. We simply make sure that for every square in the promotion path that is attacked by the opponent, we also have a piece defending that square. There are some cases where this might not be the same as being able to advance safely (according to SEE), but they are rather unusual (two knights attacking, one queen defending).

There is also one more slight difference here: in Fruit, to be a free passer, all of the above conditions must apply. In Rybka, we break the conditions down and award partial bonuses if only some of the conditions are met.

Fruit	Rybka

`if (board->piece_size[def] <= 1`
`&& (unstoppable_passer(board,sq,att) ||`
`king_passer(board,sq,att))) {`
`delta += UnstoppablePasser;`
	`if ((Board.pieces[BQ] | Board.pieces[BR] |`
	`Board.pieces[BB] | Board.pieces[BN]) == 0) {`
	`if (white_unstoppable_passer(square) ||`
	`white_king_passer(square))`
	`endgame += UnstoppablePasser;`
	`} else {`
	`if ((mob & Board.pieces[White]) == 0)`

} else if (free_passer(board,sq,att)) { delta += FreePasser; }	endgame += PassedUnblockedOwn[rank]; if ((mob & Board.pieces[Black]) == 0) endgame += PassedUnblockedOpp[rank]; if (((~mob_w) & mob & mob_b) == 0) endgame += PassedFree[rank]; }

King Distance

Both Rybka and Fruit now apply a bonus based on the distances of both kings to the square in front of the pawn. These bonuses are applying the same way, two bonuses, one for each king, simply multiplied by the distance of that king to the square in front of the pawn. The bonuses in Rybka are also based on the rank of the pawn.

Fruit	Rybka
delta -= pawn_att_dist(sq,KING_POS(board,att),att) * AttackerDistance; delta += pawn_def_dist(sq,KING_POS(board,def),att) * DefenderDistance;	endgame -= pawn_att_dist(square,wking_square,White) * PassedAttDistance[rank]; endgame += pawn_def_dist(square,bking_square,White) * PassedDefDistance[rank];

Values

If the above mentioned similarities in the evaluations were all that were there, this might be simply a coincidence. The exact same set of terms are used, and the same method of accumulating opening and endgame scores is used (interpolating between maximum and minimum based on rank, fixed score for opening, bonuses increase maximum endgame score). The free passer bonuses are separated, though, and the semantics for adding bonuses are changed (albeit into a mathematically equivalent method). But we haven't yet looked at the values for the pawns. As discussed above, Fruit's bonuses are based on the `Bonus` array, with values `{0..., 26, 77, 154, 256, 0}`, where rank 4 is 26, 5 is 77, etc. Once we look at Rybka's values, we see that they are based on the same Bonus array, and are simply precalculated outputs of the `quad()` function. Rybka's values and their Fruit equivalents (see the simplified Fruit code above) are shown below.

Also, note that in the Rybka code, the equivalent rank 8 value of the Bonus array is 256 (like rank 7) instead of 0 as in Fruit. This difference is completely meaningless however, since there can never be a pawn on the 8th rank.

Rybka	Fruit Equivalent
int PassedOpening[8] = { 0, 0, 0, 489, 1450, 2900, 4821, 4821 }; int PassedEndgame[8] = { 146, 146, 146, 336, 709, 1273, 2020, 2020 }; int PassedUnblockedOwn[8] = { 0, 0, 0, 26, 78, 157, 262, 262 }; int PassedUnblockedOpp[8] = { 0, 0, 0, 133, 394, 788, 1311, 1311 }; int PassedFree[8] = { 0, 0, 0, 101, 300, 601, 1000, 1000 }; int PassedAttDistance[8] = { 0, 0, 0, 66, 195, 391, 650, 650 }; int PassedDefDistance[8] = { 0, 0, 0, 131, 389, 779, 1295, 1295 };	int PassedOpeningMin = 0; int PassedOpeningMax = 4821; int PassedEndgameMin = 146; int PassedEndgameMax = 2020; int PassedUnblockedOwnMax = 262; int PassedUnblockedOppMax = 1311; int FreePasserMax = 1000; int AttackerDistanceMax = 650; int DefenderDistanceMax = 1295;

We also need to take a look at the candidate bonus. This bonus is done statically and stored in the pawn hash table, as discussed here, but we haven't looked at the values yet. We see that in Fruit candidates are scored using the same `quad()` function. And sure enough, Rybka's scores are based on the same array.

Rybka	Fruit Equivalent

`int CandidateOpening[8] = { 0, 0, 0, 382, 1131, 2263, 3763, 3763 };`
`int CandidateEndgame[8] = { 18, 18, 18, 181, 501, 985, 1626, 1626 };`
	`int CandidateOpeningMin = 0;`
	`int CandidateOpeningMax = 3763;`
	
	`int CandidateEndgameMin = 18;`
	`int CandidateEndgameMax = 1626;`

Material

The material tables in Rybka were one of the more interesting features introduced. Their implementation was a new way to evaluate material imbalances. The indexing and evaluations in the table seem to be unique, but there are some very interesting similarities in the information stored in the table with Fruit.

Structure

Rybka's material tables are implemented as a massive data structure that is indexed by the count of every piece on the board. The count of each piece is limited to a reasonable maximum, that can only be exceeded by promotions. This is done to keep the table a reasonable size. Pawns have a count from 0-8, minors and rooks have a count from 0-2, and queens are from 0-1. The total size of the table is thus 9*9*3*3*3*3*3*3*2*2 entries. The table is indexed in a sort of split base, with the pawn counts as the most significant indices. This means that while positions with, say, 4 queens will not overflow the index (but will point to an entry with an incorrect material configuration). The evaluation for a material configuration is stored as a 32-bit integer, which is added to the material balance determined with a sum of piece values.

In addition to the material values, Rybka keeps flags for certain situations in the table, as well as a phase value. One flag, the flag for lazy evaluation, is only in Rybka (Fruit has no lazy eval). All of the other flags come directly from Fruit.

The structure of the material table (at the source code level) isn't certain. It seems likely, based on the disassembly, that the data type is something like this:

Rybka Material Table Structure

```
struct {
 unsigned char flags;
 unsigned char cflags;
 unsigned char phase;
 bool lazy_eval;
 int mat_value;
};
```

The use of unsigned char and bool for phase and lazy eval are quite likely, because of their use: the assembly code uses the byte registers for dealing with them (al, bl, etc.). Phase is used only after zero-extending from a byte register to an int (hence the unsigned). Lazy eval is most likely a bool in the source because it takes the value 0 or 1, whereas the other flags use other bits, and it is not in contiguous memory with the flags (the phase field separates them).

The flags fields are unclear, though. In Fruit, there are two sets of flags: color dependent and not. The color dependent flags are stored in a two-element array (`cflags`) and the others are stored in one element (`flags`). Each is 8 bits, giving 8 bits total. It is at least clear that flags and cflags are stored in separate fields in Rybka--they are accessed in the assembly using `byte ptr` semantics, with cflags taken from the stack address of flags+1.

It seems that cflags is not an array though. It has two flags, MatWhiteKingFlag and MatBlackKing flag that are bits 0x08 and 0x80 respectively. The same flag is in Fruit, MatKingFlag, used with bit 0x08 (1 << 3). This is stored in cflags[color], to indicate the flag for both colors. In Rybka's material structure, it is as if it had the same array but with 4-bit bitfields instead of bytes (though it is not possible in C to have arrays of bitfields)--this would put the flags in the same bits that they are now. This compression of two bytes to one byte was most likely done so that each material table entry would be 8 bytes long. The use of 0x08 for a king safety flag in both programs is certainly interesting, though. The exact usage of the flags are dicussed below.

Also, for the one flag used in Rybka in the color-independent field, DrawBishopFlag, it is stored as bit 0x80. However, Rybka's code only tests if the flags field is nonzero, so the exact value is irrelevant. In Fruit, the same flag is in bit 0x02 (1 << 1).

Flags

Below, I compare the different material flags used in both Rybka and Fruit. I will note that all of the formulae for Rybka's flags have been decoded--since the material table is a large constant array in the Rybka executable, the code to set the flags is not there. The formulae are found by analyzing the pattern of when it appears in the material table.
There are a set of flags in Fruit that are not in Rybka. All of these (DrawNodeFlag, MatRookPawnFlag, MatBishopFlag, and MatKnightFlag) are not included in Rybka because it does not have any separate endgame knowledge, which is the purpose of all of these flags in Fruit. Rybka has all other flags that are in Fruit, and also an additional lazy evaluation flag. Fruit does not have lazy evaluation, so there is no flag in it.

MatKingFlag

Fruit stores a flag for each color for whether king safety will be evaluated. This is stored as 0x08 in the cflags array in the material table (see above). The formula for this table is shown below. In Rybka, the exact same formula is used--if the enemy has a queen and at least two pieces total, king safety is evaluated for that side. Note also that cflags is a single byte, not a two-byte array as in Fruit (see above, again).

MatKingFlag in Fruit	MatKingFlag in Rybka

`const int MatKingFlag = 1 << 3;`
 	
`if (bq >= 1 && bq+br+bb+bn >= 2)`
` cflags[White] |= MatKingFlag;`
`if (wq >= 1 && wq+wr+wb+wn >= 2)`
` cflags[Black] |= MatKingFlag;`
	`const int MatWhiteKingFlag = 1 << 3;`
	`const int MatBlackKingFlag = 1 << 7;`
	
	`if (bq >= 1 && bq+br+bb+bn >= 2)`
	` cflags |= MatWhiteKingFlag;`
	`if (wq >= 1 && wq+wr+wb+wn >= 2)`
	` cflags |= MatBlackKingFlag;`

DrawBishopFlag

Fruit and Rybka store a flag in their material tables for signifying the possibility of an opposite-color bishop endgame, which is generally drawish. The flag has the exact same formula in both programs: there must be only bishops and pawns, each side must have exactly one bishop, and the difference in the number of pawns of each side cannot be more than two.

DrawBishopFlag in Fruit	DrawBishopFlag in Rybka

`const int DrawBishopFlag = 1 << 1;`
 	
`if (wq+wr+wn == 0 && bq+br+bn == 0) {`
` if (wb == 1 && bb == 1) {`
` if (wp-bp >= -2 && wp-bp <= +2) {`
` flags |= DrawBishopFlag;`
` }`
` }`
`}`
	`const int DrawBishopFlag = 1 << 7;`
	
	`if (wq+wr+wn == 0 && bq+br+bn == 0) {`
	` if (wb == 1 && bb == 1) {`
	` if (wp-bp >= -2 && wp-bp <= +2) {`
	` flags |= DrawBishopFlag;`
	` }`
	` }`
	`}`

The usage of these flags is just as interesting: at the very end of the evaluation, after the total score is computed, the flag is checked. Since both programs do not distinguish the color of the bishops in the material table, the flag only indicates whether an OCB ending is possible. The color of the bishops must still be checked. The actual check is done in different ways because Rybka is in bitboards, but the test has the same meaning. In Fruit, if it is really an OCB ending, the `mul` value is set to 8 for each side (provided a draw recognizer has not already marked this as a drawish ending). After this check, Fruit multiplies the score by mul[color]/16, with the color depending on which side is ahead. If both sides have a value of 8, as is the case when there is not a draw recognition, this has the effect of dividing the score by two, bringing it closer to the draw value, 0. In Rybka, there is no mul value, as there aren't any draw recognizers. But we see that in the case of an OCB ending, it does the same thing as Fruit: divide the score by two.

DrawBishopFlag usage in Fruit	DrawBishopFlag usage in Rybka

```
if ((mat_info->flags & DrawBishopFlag) != 0) {
 wb = board->piece[White][1];
 bb = board->piece[Black][1];
 if (SQUARE_COLOUR(wb) != SQUARE_COLOUR(bb)) {
 if (mul[White] == 16) mul[White] = 8;
 if (mul[Black] == 16) mul[Black] = 8;
 }
}
```
	```
	if (flags & DrawBishopFlag) {
	 mask = Board.pieces[BB] | Board.pieces[WB];
	 if ((mask & MaskLightSquares) && (mask &
	MaskDarkSquares)) {
	 opening = opening / 2;
	 endgame = endgame / 2;
	 }
	}
	```

Lazy Evaluation

In addition to the flags discussed above, Rybka stores a boolean flag for whether to perform lazy evaluation or not. Rybka has an extremely aggressive lazy eval--if the material difference (not including the material table offset) is beyond bounds set at the root based on previous iterations, the evaluation is based only on material (this time including the material table offset). In addition to these cases, there are a set of material configurations for which lazy evaluation (material only) is performed unconditionally. For instance, in a KRR vs KQN ending, Rybka does absolutely no evaluation beyond material--it simply returns a constant value, regardless of previous search values or the position of pieces. The pattern of material configurations which have this flag set is not very clear. There are 1106 such configurations (though due to symmetry there are only 553 unique ones). Each of these configurations also has in common that they are not equal (the material is imbalanced), but the difference in material value is not very large (the only configurations with more than 4 pawns difference are KNN vs K and KNN vs KP). Beyond that, though, it's not very clear. Perhaps these configurations were harvested from a collection of games and found to have some property. There are certainly too many configurations, including very obscure ones (such as KQRBPPPPP vs KQBBNN), for this to have been done by hand.

However, there is a very serious bug in Rybka with regards to lazy evaluation. The upper and lower bounds are set to the root score at the end of every iteration that is at least 6 plies. However, Rybka deals with two different scales of evaluation: units of a centipawn and units of 1/32 of a centipawn. In this case, the two values are mixed up: Rybka's search value is in centipawns, but it sets the lazy eval as if this value were in 1/32 centipawn units. Thus, every evaluation (that happens to be less than 32 pawns in either direction, i.e. always) will cause the lazy evaluation bounds to be set based on a score of 0. This means that if the root score (before dividing by 3399) is >0, the bounds are set to -3 and 4, and if the score is <0, the bounds are set to -4 and 3. Every single position with a score outside of these bounds is lazily evaluated, which means that once the score is in this range, Rybka effectively switches to material-only evaluation.

Phase

One of the more unique aspects of the Fruit evaluation is that it calculates two different scores, for opening and endgame, and interpolates between the two based on the phase of the game (which is calculated from the material left on the board). This was quite uncommon when Fruit first appeared (if it was used elsewhere at all), though in the meantime many other engines have begun to use this strategy. It is interesting that Rybka uses the same approach (with one interesting modification), though it is not necessarily evidence of any wrongdoing. Looking at the phase value that is used to interpolate between the two values, however, it is very clear that Rybka has copied Fruit's values.

Both Fruit and Rybka store the phase value in the material table. Fruit's formula is pretty simple: for the opening, a phase of 0 is used, and for the endgame, 256. This is calculated by taking phase values for each piece (pawns do not count, minors count for 1, rooks for 2, and queens 4). The total of these values is subtracted from TotalPhase (which is 24). This is then expanded into the 0-256 range with a simple proportionality constant.

Phase in Fruit	Phase in Rybka
	`static const int PawnPhase = 0;`

```
static const int PawnPhase = 0;
static const int KnightPhase = 1;
static const int BishopPhase = 1;
static const int RookPhase = 2;
static const int QueenPhase = 4;

static const int TotalPhase = PawnPhase * 16 +
 KnightPhase * 4 + BishopPhase * 4 +
 RookPhase * 4 + QueenPhase * 2;

...

phase = TotalPhase;

phase -= wp * PawnPhase;
phase -= wn * KnightPhase;
phase -= wb * BishopPhase;
phase -= wr * RookPhase;
phase -= wq * QueenPhase;

phase -= bp * PawnPhase;
phase -= bn * KnightPhase;
phase -= bb * BishopPhase;
phase -= br * RookPhase;
phase -= bq * QueenPhase;

if (phase < 0) phase = 0;

phase = (phase * 256 + (TotalPhase / 2)) /
TotalPhase;
```

```
static const int KnightPhase = 1;
static const int BishopPhase = 1;
static const int RookPhase = 2;
static const int QueenPhase = 4;

static const int TotalPhase = PawnPhase * 16 +
 KnightPhase * 4 + BishopPhase * 4 +
 RookPhase * 4 + QueenPhase * 2;

...

phase = TotalPhase;

phase -= wp * PawnPhase;
phase -= wn * KnightPhase;
phase -= wb * BishopPhase;
phase -= wr * RookPhase;
phase -= wq * QueenPhase;

phase -= bp * PawnPhase;
phase -= bn * KnightPhase;
phase -= bb * BishopPhase;
phase -= br * RookPhase;
phase -= bq * QueenPhase;

if (phase < 0) phase = 0;

phase = (phase * 256 + (TotalPhase / 2)) /
TotalPhase;

phase /= 4;
```

Rybka has the same formula as Fruit. There is one important difference though: in order for the value to fit into the one-byte field in the material table (which has a range of only 0-255, instead of 0-256), it is divided by 4, bringing the range from 0 to 65. There is not much loss here, since the values are extrapolated from only 25 different phase values. It is interesting to note, however, that only 25 of the values are ever possible. Rybka could have simply stored the 0-25 phase without extrapolating to a larger range. Since the phase is used to index a table (see below), this means that there are 40*2 entries which are never accessed in this table. In my opinion, this makes it clear that the original code wasn't understood fully.

In Rybka, the final interpolation between opening and endgame scores is done using a table, phase_value[65][2]. The opening value is multiplied by phase_value[phase][0], the endgame value is multiplied by phase_value[phase][1], and these are added together. This is then divided by 256*32--the sum of each phase_value for opening and endgame is around 256, and Rybka evaluates with a base of 32 units per centipawn (with the pawn actually worth 3399, about 106 centipawns). Each of these values (256 and 32) are confirmed by looking at other places in the eval: when setting the lazy eval, Rybka multiplies by 256 and divides by the sum of the two phase values. When returning the lazy eval, it takes the material difference multiplied by 3399, adds the material table offset, and divides by 32.

The phase_value table has values which are not quite simple, but when divided into three sections of phases (0-12, 13-51, 52-64), the values can be quite closely described by quadratic equations. This gives six total equations.

Pawn Evaluation

Rybka's pawn evaluation is very simple. It is again, virtually identical to Fruit's. The bitboard structure allows for a much more efficient calculation though. The comparison between them is also very simple.

Pawn Hash Table

First, we will compare the entries for the pawn hash table. Both entries have a 32-bit hashkey, two signed 16-bit scores for opening and endgame, two 8-bit file-wise bitmasks for passed pawn files, and a 16 bit pad. In Fruit, the rest is used by 16 bits of flags (of which only 2 bits are actually used) and two 8-bit squares that are used for draw recognition. Rybka does not have draw recognition (it is replaced with the material table), so this information is useless. In Rybka, those 4 bytes are replaced by 12 bytes grouped into 3x2 16-bit scores. These scores are cached king shelter scores, which are discussed here.

Terms

The evaluation terms discussed below use a side-by-side comparison as always with Fruit and approximate Rybka code. See also the decompilation notes. In Fruit's pawn structure evaluation the patterns are computed first (doubled, isolated, etc.), and the scores are computed afterwards. The Rybka decompilation uses a more compact style, which likely matches the original Rybka code closer (based on the assembly output). Also, virtually all of Rybka has white and black coded separately. The white code is used here for the examples. Also, note the similarity of giving an extra penalty, only in the opening, for some features if the pawn is on an open file. The endgame score subtraction for backward pawns is made outside of the if-block in Rybka, but the meaning is still identical. The change is almost certainly made by the optimizer anyways, since for isolated pawns the subtraction is inside the if-block.

Also, it should be noted that each evaluation uses the exact same terms in the exact same order: doubled, isolated, backwards, passers, candidates. I want to stress that all of the differences shown below are very minor implementational details, that would be quite natural given a translation of Fruit to bitboards. Overall, the pawn evaluations of each program are essentially identical.

Doubled Pawns

Doubled pawns are the first and simplest pattern. In Fruit, we look behind the given pawn for a friendly pawn. In Rybka, we look ahead. These are of course equivalent. Rybka also has a score of zero for doubled pawns in the opening.

Fruit	Rybka

`if ((board->pawn_file[me][file] & BitLT[rank]) != 0)`
`        doubled = true;`
 	
`...`
 	
`if (doubled) {`
`        opening[me] -= DoubledOpening;`
`        endgame[me] -= DoubledEndgame;`
`}`
	`if (MaskPawnDoubled[square] & Board.pieces[WP])`
	`        endgame -= DoubledEndgame;`

Isolated Pawns

Isolated pawns come next. In Fruit, the variable t1 represents the bitwise OR of the rank-wise bitmasks of friendly pawns on adjacent files. So if t1==0, that means that no pawns are on either of the adjacent files to this pawn. Rybka's

`MaskPawnIsolated` works the same way.

Fruit	Rybka

if (t1 == 0)
 isolated = true;
 	
...
 	
if (isolated) {
 if (open) {
 opening[me] -= IsolatedOpeningOpen;
 endgame[me] -= IsolatedEndgame;
 } else {
 opening[me] -= IsolatedOpening;
 endgame[me] -= IsolatedEndgame;
 }
}
	if ((MaskPawnIsolated[square] &
	Board.pieces[WP]) == 0) {
	 if (open_file) {
	 opening -= IsolatedOpeningOpen;
	 endgame -= IsolatedEndgame;
	 } else {
	 opening -= IsolatedOpening;
	 endgame -= IsolatedEndgame;
	 }
	}

Backward Pawns

Backward pawns are next, and are one of the more complicated pawn terms. t1 is as discussed above. t2 is the rank-wise bitmask of all pawns on the same file as the pawn. First, we test whether the pawn is behind all friendly pawns that might be on adjacent files: `t1 & BitLE[rank]` in Fruit, `(MaskPawnProtectedW[square] & Board.pieces[WP]) == 0` in Rybka. Next, we test if the pawn is "really backward". This basically means that the pawn can't advance one square to meet a friendly pawn. We also have to check for pawns on the second rank, because they could possibly advance twice to meet another pawn. We see if there is a pawn blocking the advance, or an opponent pawn attacking the advance square. In Rybka this is a bit different, because any pawn (not just those on the second rank) can advance twice to escape backwardness, and because it is not checked whether there is another pawn blocking the pawn from advancing.

Fruit	Rybka

if ((t1 & BitLE[rank]) == 0) {
backward = true;
// really backward?
if ((t1 & BitRank1[rank]) != 0) {
ASSERT(rank+2<=Rank8);
if (((t2 & BitRank1[rank])
| ((BitRev[board->pawn_file[opp][file-1]] |
BitRev[board->pawn_file[opp][file+1]]) &
BitRank2[rank])) == 0) {
backward = false;
}
} else if (rank == Rank2 && ((t1 &
BitEQ[rank+2]) != 0)) {
ASSERT(rank+3<=Rank8);
if (((t2 & BitRank2[rank])
| ((BitRev[board->pawn_file[opp][file-1]] |
BitRev[board->pawn_file[opp][file+1]]) &
BitRank3[rank])) == 0) {
backward = false;
}
}
}
 	
...
 	
 if (backward) {
if (open) {
opening[me] -= BackwardOpeningOpen;
endgame[me] -= BackwardEndgame;
} else {
opening[me] -= BackwardOpening;
endgame[me] -= BackwardEndgame;
}
}
	if ((MaskPawnProtectedW[square] &
	Board.pieces[WP]) == 0) {
	if ((MaskPawnAttacksW1[square] &
	Board.pieces[BP]) ||
	((MaskPawnAttacksW2[square] & Board.pieces[BP])
	&&
	((MaskPawnAttacks[White][square] &
	Board.pieces[WP])==0))) {
	if (open_file)
	opening -= BackwardOpeningOpen;
	else
	opening -= BackwardOpening;
	endgame -= BackwardEndgame;
	}
	}

Passed Pawns

Passed pawns are simply detected at this point, and follow the standard definition. The dynamic evaluation of pawns is discussed here. For now, we store an 8-bit mask for each side with the files containing passed pawns.

Fruit	Rybka

`if (((BitRev[board->pawn_file[opp][file-1]] |`
`BitRev[board->pawn_file[opp][file+1]]) &`
`BitGT[rank]) == 0) {`
`passed = true;`
`passed_bits[me] |= BIT(file);`
`}`
	`if ((MaskPawnPassedW[square] &`
	`Board.pieces[BP]) == 0)`
	`wp_pass_file |= PawnPassedFile[file];`

Candidate Passed Pawns

Candidate passed pawns have a similar definition in both programs. The pawn must be on an open file. Then we take the count of all defender pawns (friendly pawns behind) and the count of all attacker pawns (enemy pawns in front). If there are an equal or greater number of defenders, the pawn is a candidate. There is an exception in Fruit though--if the pawn has enough defenders, we also check the count of direct defenders and attackers, that is, pawns that are already attacking our pawn. The number of direct defenders must also be greater or equal to the number of direct attackers. While the definition is almost identical, the scores here are where we get an early glimpse of the real similarities. The scoring is discussed more here.

Fruit	Rybka

`n = 0;`
`n += BIT_COUNT(board->pawn_file[me][file-`
`1]&BitLE[rank]);`
`n += BIT_COUNT(board-`
`>pawn_file[me][file+1]&BitLE[rank]);`
`n -= BIT_COUNT(BitRev[board->pawn_file[opp][file-`
`1]]&BitGT[rank]);`
`n -= BIT_COUNT(BitRev[board-`
`>pawn_file[opp][file+1]]&BitGT[rank]);`
`if (n >= 0) {`
`// safe?`
`n = 0;`
`n += BIT_COUNT(board->pawn_file[me][file-`
`1]&BitEQ[rank-1]);`
`n += BIT_COUNT(board-`
`>pawn_file[me][file+1]&BitEQ[rank-1]);`
`n -= BIT_COUNT(BitRev[board->pawn_file[opp][file-`
`1]]&BitEQ[rank+1]);`
`n -= BIT_COUNT(BitRev[board-`
`>pawn_file[opp][file+1]]&BitEQ[rank+1]);`
`if (n >= 0) candidate = true;`
`...`
 	
`if (candidate) {`
`opening[me] +=`
`quad(CandidateOpeningMin,CandidateOpeningMax,rank);`
`endgame[me] +=`
`quad(CandidateEndgameMin,CandidateEndgameMax,rank);`
`}`
	`if (open_file) {`
	`mask1 = MaskPawnProtectedW[square] &`
	`Board.pieces[WP];`
	`mask2 = MaskPawnPassedW[square] &`
	`Board.pieces[BP];`
	`if (popcnt(mask1) >= popcnt(mask2)) {`
	`opening += CandidateOpening[rank];`
	`endgame += CandidateEndgame[rank];`
	`}`
	`}`

Piece Evaluation

Rybka and Fruit evaluate pieces next. This evaluation is very simple, primarily based on mobility. There are a few more bonuses besides that. Firstly, we will examine the mobility evaluation of both programs to show their equivalence.

Mobility

The mobility calculations of Fruit and Rybka seem different, but Rybka's turns out to be a simple bitboard translation of Fruit's.

Fruit's mobility is based on the MobUnit[][] array. It is indexed by the color of the piece we're evaluating the mobility for, and then by the piece type of the piece that's being attacked. While this seems complicated, the initialization turns out to be very simple: one point is given for attacking an empty piece or an opponent piece, and no points are given for attacking friendly pieces. This point total is added to an offset and is then multiplied by a weight. Here is the bishop mobility to illustrate this point:

Bishop Mobility in Fruit

```
mob = -BishopUnit;

for (to = from-17; capture=board->square[to], THROUGH(capture); to -= 17) mob += MobMove;
mob += unit[capture];

// Other directions...

op[me] += mob * BishopMobOpening;
eg[me] += mob * BishopMobEndgame;
```

Note also that Fruit has a constant added to each piece (BishopUnit for bishops, etc.). Since this is just a constant, it can be added into the piece value (by subtracting BishopUnit*BishopMobOpening for opening, etc.), thus it is not important to the semantics of the code.
This type of calculation is very easily expressed in bitboards using a mask and a population count. Rybka's mobility evaluation is indeed a direct translation of Fruit's code to bitboards. The set of squares attacked by the piece (bishop in this case), but which are not friendly pieces, are given one point each, counted using the `popcnt()` function. Note that this number is the same as "mob" as used in Fruit. This is then multiplied by the weights for opening and endgame for each piece and added to the totals.
The attack bitboards that are calculated for mobility in Rybka are also used for King Safety evaluation

Bishop Mobility in Rybka

```
attacks = bishop_attacks(square);

// evaluate king safety here...

mob = popcnt(attacks & ~own_pieces);
opening += mob * BishopMobOpening;
endgame += mob * BishopMobEndgame;
```

More Piece Evaluation

Besides mobility, Fruit and Rybka only have a few basic terms for piece evaluation. An explanation of the bonuses for each piece follows.

Minors

In both Rybka and Fruit, knights and bishops are evaluated only based on mobility. No other terms are used.

Rooks

In Fruit and Rybka, there are two main rook bonuses: open file and seventh rank. Open files are fairly simple, but have a rather uncommon formulation that Fruit and Rybka share. Also, like mobility above, Fruit adds in a constant for every rook (to balance the open file scores between positive and negative). This can be added in to the piece score, and can be ignored for the analysis here.

In both Rybka and Fruit, we start by checking if there is a friendly pawn on the same file. In Rybka, however, we only check for pawns in front of the rook. This is a "semi-open" file. If there aren't any, we then check for an enemy pawn on the same file.

Fruit

```
op[me] -= RookOpenFileOpening / 2;
eg[me] -= RookOpenFileEndgame / 2;

rook_file = SQUARE_FILE(from);
if (board->pawn_file[me][rook_file] == 0) {
 op[me] += RookSemiOpenFileOpening;
 eg[me] += RookSemiOpenFileEndgame;

 if (board->pawn_file[opp][rook_file] == 0) {
 op[me] += RookOpenFileOpening -
 RookSemiOpenFileOpening;
 eg[me] += RookOpenFileEndgame -
 RookSemiOpenFileEndgame;
 }

 ...
}
```

Rybka

```
static const int RookSemiOpenFileOpening = 64;
static const int RookSemiOpenFileEndgame = 256;
static const int RookOpenFileOpening = 1035;
static const int RookOpenFileEndgame = 428;

file_bb = mask_open_file_w[square];
if ((Board.pieces[WP] & file_bb) == 0) {
 opening += RookSemiOpenFileOpening;
 endgame += RookSemiOpenFileEndgame;

 if ((Board.pieces[BP] & file_bb) == 0) {
 opening += RookOpenFileOpening -
 RookSemiOpenFileOpening;
 endgame += RookOpenFileEndgame -
 RookSemiOpenFileEndgame;
 }

 ...
}
```

To extend the open file evaluation, we check if the open file is one of the three surrounding files of the opponent king. In Fruit, this is done by checking if the file-distance between the rook and king is less than or equal to one. In Rybka, the same calculation is done with a bitboard mask, by ANDing the file of the rook with the opponent king's attack set. If the rook is on the same file as the king, we add an additional bonus.

Fruit

```
if ((mat_info->cflags[opp] & MatKingFlag) != 0)
{
 king = KING_POS(board,opp);
 king_file = SQUARE_FILE(king);
 delta = abs(rook_file-king_file);
 if (delta <= 1) {
 op[me] += RookSemiKingFileOpening;
 if (delta == 0)
 op[me] += RookKingFileOpening -
RookSemiKingFileOpening;
 }
}
```

Rybka

```
static const int RookSemiKingFileOpening = 121;
static const int RookKingFileOpening = 974;


if ((flags & MatBlackKingFlag) && (bking_moves
& file_bb)) {
 opening += RookSemiKingFileOpening;
 if (Board.pieces[BK] & file_bb)
 opening += RookKingFileOpening -
RookSemiKingFileOpening;
}
```

Finally, we check for a rook on the seventh rank. In order for the rook to get the bonus, either the opponent must have pawns on the second rank, or their king on the first rank.

Fruit	Rybka

```
if (PAWN_RANK(from,me) == Rank7) {
 if ((pawn_info->flags[opp] & BackRankFlag) != 0
 || PAWN_RANK(KING_POS(board,opp),me) == Rank8) {
 op[me] += Rook7thOpening;
 eg[me] += Rook7thEndgame;
 }
}
```
	```
	if (RankOf(square) == Rank7) {
	 if ((Board.pieces[BP] & MaskRank7) ||
	 (Board.pieces[BK] & MaskRank8)) {
	 endgame += Rook7thEndgame;
	 opening += Rook7thOpening;
	 }
	}
	```

Queens

Besides mobility, the only evaluation term for queens is the seventh rank bonus. This bonus is calculated in the exact same way as for rooks above, but with a different value.

Conclusion

From looking at the piece evaluation of both engines, we find that they are almost identical. As with most evaluation terms, Rybka's weights have been tuned differently. The only other difference in the piece evaluation is that, in Rybka, open files for rooks are based only on the pawns in front of the rook. Open files for rooks in Fruit are based on every pawn on the file. Of course, this difference is fairly trivial, and will not make a difference most of the time.

King Evaluation

Rybka's and Fruit's king evaluation are both very similar. Rybka's is a heavily optimized version though.

Using King Safety

First, we have to actually determine whether to use king safety or not. In both programs, we keep a flag for each side in the material table. This flag has the same meaning in both programs: if the opponent has a queen, and at least one other piece (rook, bishop, or knight), then we must calculate king safety for that side.

Attacks

For the main part of the king safety evaluation, we calculate which pieces are attacking the king and its surrounding area. This requires generating the attacks for each piece. Whereas in Fruit these attacks are calculated in anindependent function, Rybka saves time by using the attacks for each piece while calculating mobility. The functions are equivalent, but Rybka spends only have the time generating attacks. Rybka has separate code for each piece as well.

The `piece_attack_king()` function in Fruit detects whether a piece attacks any of the (up to) 8 squares surrounding the king, while `bking_area` in Rybka is a bitboard of those surrounding squares. If the attacks bitboard intersects with this bitboard, then the piece attacks the king area. Note that in both programs, only the surrounding squares count, not the actual square of the king.

We keep two counters for this calculation: the number of pieces that attack the opponent king's area, and a sum of weights of those pieces. The weights are different in Rybka of course, but both programs have zero weight for pawns.

Fruit	Rybka (knights)

`if (piece_attack_king(board,piece,from,king)) {`
`piece_nb++;`
`attack_tot += KingAttackUnit[piece];`
`}`
	`attacks = knight_attacks(square);`
	`mob_w |= attacks;`
	`if (attacks & bking_area) {`
	`piece_nb++;`
	`attack_tot += KingAttackUnit[Knight];`
	`}`
	`// mobility calculation here`

Once we've gone through all the piece attacks, we need to get our final score. This is done with an array lookup of weights indexed by the number of pieces attacking the king, multiplied by the sum of the weights of those pieces. Fruit keeps a separate factor `KingAttackOpening` that is also multiplied in. Rybka effectively keeps this factor inside the constant `KingAttackUnit` table. This should be seen as simply a speed optimization.

Fruit	Rybka

`op[colour] -= (attack_tot * KingAttackOpening *`
`KingAttackWeight[piece_nb]) / 256;`
	`opening -= (KingAttackWeight[piece_nb] *`
	`attack_tot) / 32;`

Shelter

After we have calculated the king attacks, we evaluate the king's shelter. The king shelter in both Rybka and Fruit measures the safety of the king position based on the positions of the pawns on the (up to) 3 files adjacent to the king. This comparison is rather tricky, since Rybka's evaluation is heavily optimized.

First, we will look at Fruit. Fruit takes three scores: the shelter score for the king's current position, and two "castling" scores. The castling scores are the shelter scores for the castling target squares if the king is able to castle in that direction. From these scores we take two penalties. The first penalty is the score for the current position, and the second penalty is the best of all three scores. This means that if the current king position has a bad shelter, but we can castle into a position with a good shelter, the penalty shouldn't be too bad. We then take the average of the two penalties to get our final shelter score.

Rybka does the same thing, but in a way that can be optimized for storage in the pawn hash table. The difference is that when we're first getting our score, our king square is generalized to either C1, E1, or G1 (or the reverse square for black). This is done with the `SquareWing` table. If the king is on the A, B, or C files. we take the score as if it was on C1; for the D and E files, E1; and for the F, G, and H files G1. The two castling scores are based on C1 and G1 of course. Since there are only three possible squares the king shelter can be evaluated for, Rybka can store the 3 values for each side in the pawn hash table. We see in the below code that Rybka's `entry->white_king_shelter[KingSide]` is equivalent to Fruit's `shelter_square(board,G1,me)`, etc.

Fruit

```
penalty_1 = shelter_square(board,KING_POS(board,me),me);
penalty_2 = penalty_1;

if ((board->flags & FlagsWhiteKingCastle) != 0) {
tmp = shelter_square(board,G1,me);
if (tmp < penalty_2) penalty_2 = tmp;
}

if ((board->flags & FlagsWhiteQueenCastle) != 0) {
tmp = shelter_square(board,B1,me);
if (tmp < penalty_2) penalty_2 = tmp;
}

penalty = (penalty_1 + penalty_2) / 2;
op[me] -= (penalty * ShelterOpening) / 256;
```

Rybka

```
wing = SquareWing[wking_square];
penalty_1 = entry->white_king_shelter[wing];
penalty_2 = penalty_1;

if ((Board.flags & FlagsWhiteKingCastle) != 0) {
tmp = entry->white_king_shelter[KingSide];
if (tmp < penalty_2) penalty_2 = tmp;
}

if ((Board.flags & FlagsWhiteQueenCastle) != 0) {
tmp = entry->white_king_shelter[QueenSide];
if (tmp < penalty_2) penalty_2 = tmp;
}

opening -= (penalty_1 + penalty_2) / 2;
```

Shelter Values

We have seen that the way Rybka and Fruit evaluate shelter for both the king position and the two castled positions, and how they are combined. Now we have to look at how they evaluate the shelter for each position.

We need to look at the three adjacent files to a square. In Fruit, for each of these files we take the furthest back pawn that is still in front of the king on that file. The penalty scales quadratically from 36 to 0 going in reverse. So a pawn on the 2nd rank gets a penalty of 0, a pawn on the 3rd rank gets a penalty of 11, 4th rank gets 20, etc. This bonus is different in Rybka, and is simply a table.

`shelter_file()` in Fruit

```
dist = BIT_FIRST(board->pawn_file[colour][file]&BitGE[rank]);

dist = Rank8 - dist;

penalty = 36 - dist * dist;
```

Rybka equivalent

```
const int shelter_value[5] = { 1121, 0, 214, 749, 915 };

if (pawns_on_file) {
 dist = min_rank;
 dist = MIN(4, min_rank);
} else
 dist = 0;

penalty = shelter_value[dist];
```

Next we have pawn storms. This is basically the same as pawn shelters, but we look at the opponent's pawns. Fruit's way of calculating it is a bit different, while Rybka uses essentially the same method as it uses for shelter files. We add the score for each file into the penalty.

`storm_file()` in Fruit	Rybka equivalent

penalty = 0;
switch (dist) {
case Rank4:
penalty = StormOpening * 1;
break;
case Rank5:
penalty = StormOpening * 3;
break;
case Rank6:
penalty = StormOpening * 6;
break;
}
	const int storm_value[5] = { 0, 0, 2334, 653,
	310 };
	
	if (pawns_on_file) {
	 dist = min_rank;
	 dist = MIN(4, min_rank);
	} else
	 dist = 0;
	
	penalty = shelter_value[dist];

We then combine the shelter scores of the three files by multiplying the center score by 2 and adding the other two. We also apply the "weak back rank" bonus if all three pawns are on the second rank (possibly exposing the king to back rank mates). Rybka does the same addition for the bonuses, and also applies the back rank bonus. The storm scores are simply added in, with no doubling in the center.

`shelter_square()` in Fruit	Rybka equivalent

penalty += shelter_file(board,file,rank,colour)
* 2;
if (file != FileA)
 penalty += shelter_file(board,file-
1,rank,colour);
if (file != FileH)
 penalty +=
shelter_file(board,file+1,rank,colour);
 	
if (penalty == 0) penalty = 11;
 	
penalty += storm_file(board,file,colour);
if (file != FileA)
 penalty += storm_file(board,file-1,colour);
if (file != FileH)
 penalty += storm_file(board,file+1,colour);
	penalty += shelter_file(board,file,rank,colour)
	* 2;
	if (file != FileA)
	 penalty += shelter_file(board,file-
	1,rank,colour);
	if (file != FileH)
	 penalty +=
	shelter_file(board,file+1,rank,colour);
	
	if (penalty == 0) penalty = 794;
	
	penalty += storm_file(board,file,colour);
	if (file != FileA)
	 penalty += storm_file(board,file-1,colour);
	if (file != FileH)
	 penalty += storm_file(board,file+1,colour);

All of this shelter evaluation code in Rybka above is an equivalent; it doesn't appear in the Rybka binary. It is there simply to illustrate what is in the precomputed tables. These precomputed tables are used during the pawn evaluation to quickly evaluate shelters. Rybka makes two bitmasks, representing the friendly and enemy pawns in front of the king. We take the 4x3 rectangle of the closest pawns in front of the king. For instance, with the king on G1, we take the pawn positions on the rectangle F5-H5-H2-F2. This creates two masks of 12 bits each to index a table of 4096 entries. In each entry we store the value computed above in Rybka's `shelter_square()`-equivalent function.

Pattern Evaluation

Fruit and Rybka both have a small pattern evaluation. These simply help prevent a few basic positional blunders that are hard to evaluate properly without special code. There are only three separate patterns, and all are present in the same form in both programs. The only functional difference between the two pattern evaluations is that Rybka does not divide the trapped bishop bonus by 2 in the case where Fruit does.

Trapped Bishop

The first part of the `eval_pattern()` function looks at trapped bishops. By Fruit's definition, a trapped bishop is simply a bishop on A6, A7, or B8 with an opponent pawn diagonally in front of it, for example a white bishop on B8 and a black pawn on C7. This is to prevent the common error like Bxa7 a6, where the bishop captures a pawn but then becomes useless. The mirror image also holds, so a white bishop on G8 with a black pawn on F7 is also trapped, and a black bishop on B1 with a white pawn on C2 is trapped. If the bishop is on A6, the penalty is halved. Rybka's trapped bishop evaluation has the same definition, with the only difference being that trapped bishops on A6/H6/A3/H3 get the full bonus instead of just half. This allows for a very quick calculation in bitboards, where all the possible squares to be trapped on are calculated simultaneously. Rybka tests for a pawn on B5, B6, or C7 with a pawn diagonally behind it (as well as the mirror images as noted) with just 3 64-bit operations.

Fruit	Rybka

```
 if ((board->square[A7] == WB && board-
>square[B6] == BP)
|| (board->square[B8] == WB && board-
>square[C7] == BP)) {
*opening -= TrappedBishop;
*endgame -= TrappedBishop;
}
if ((board->square[H7] == WB && board-
>square[G6] == BP)
|| (board->square[G8] == WB && board-
>square[F7] == BP)) {
*opening -= TrappedBishop;
*endgame -= TrappedBishop;
}
 if (board->square[A6] == WB && board-
>square[B5] == BP) {
*opening -= TrappedBishop / 2;
*endgame -= TrappedBishop / 2;
}
if (board->square[H6] == WB && board-
>square[G5] == BP) {
*opening -= TrappedBishop / 2;
*endgame -= TrappedBishop / 2;
}
```
	```
	if (((Board.pieces[WB] >> 7) & Board.pieces[BP]
	& MaskB5B6C7) ||
	((Board.pieces[WB] >> 9) & Board.pieces[BP] &
	MaskG5G6F7)) {
	 opening -= TrappedBishop;
	 endgame -= TrappedBishop;
	}
	```

Blocked Bishop

Next we test for a blocked bishop. This is where a bishop is still on its initial square and is prevented from moving by a pawn on D2/E2, and with a piece blocking this pawn from advancing. This calculation is made in a very straightforward way in both Rybka and Fruit. Rybka uses bitboard masks to test for squares instead of using its 64-element `sq` array. Since this isn't a parallel operation, the bitboards do not speed up the calculation.

Fruit	Rybka

```
if (board->square[D2] == WP && board-
>square[D3] != Empty &&
```
	```
	if ((Board.pieces[WB] & MaskC1) &&
	(Board.pieces[WP] & MaskD2) &&
	```

```
 board->square[C1] == WB) {
 *opening -= BlockedBishop;
}
```
	```
	(Board->occupied & MaskD3))
	 opening -= BlockedBishop;
	```

Blocked Rook

Finally we look for a blocked rook. This pattern prevents a king moving into a wing without castling, thereby trapping the rook to the side of the king. This test is done with a quick approximation, by only checking the position of the king and rook, and not the pawns. Fruit and Rybka have functionally identical code here too, and again Rybka's bitboard structure allows for the quick parallel testing of the king and rook positions.

Fruit	Rybka

```
if ((board->square[C1] == WK || board-
>square[B1] == WK)
&& (board->square[A1] == WR || board-
>square[A2] == WR ||
board->square[B1] == WR)) {
*opening -= BlockedRook;
}
```
	```
	if ((Board.pieces[WR] & MaskA1B1A2) &&
	(Board.pieces[WK] & MaskB1C1))
	*opening -= BlockedRook;
	```