Cilck on the picture to return to the main menu.

This document is the result of one year of counter studying the ICGA-Rybka investigation and is the collected work of several chess programmers although expressed in my own words. It's my wish and advice that in future events like these the presented evidence will be subject to more scrutiny, research and discussion than that what happened during the ICGA Panel deliberations.

A further well meant advice would be to install 2-3 qualified people who deliberately will play devils advocate until the last bug is removed. This to avoid groupthinking and tunnel visions.

This document (also available as PDF) is divided into:

1. Introduction - Plagiarism
2. Analysis document Zach Wegner
3. Analysis document Mark Watkins
4. External evidence
5. Historic evidence
6. Additional evidence of Rybka originality NEW
7. An experiment NEW

Special thanks

To Andrew Dalke for his insight knowledge on copyright infringement.

Chris Whittington, Dann Corbit, Sven Schuele, Miguel A. Ballicora, Marcel van Kervinck and Ronald de Man for their insightful comments and analysis. Although these gentlemen have a view of their own regarding the Rybka-Fruit case without their help

I could not have compiled this document.

A special thanks to Richard Vida for further decompilation of those parts of Rybka 1.0 beta which were not investigated.

Also a word of thanks to my opponents in the Rybka-Fruit debate (Mark Watkins, Zach Wegner and Robert Hyatt) for their patience to answer most of my painstaking and persistent questions.

Apologies in advance for the bad formatting at times, it's related to the tool I used to convert Wegner's PDF to HTML.

Ed Schroder
September 2012

Introduction
plagiarism

Andrew Dalke is a software specialist who contacted me about the ICGA-Rybka case. He discovered an error in Watkins document nobody until then noticed.

Andrew wrote me his findings.

Indeed, their reports show that there are extensive differences between the two implementations, like Rybka's use of a bitboard. They argue that those changes are mechanical transformations of the Fruit implementation, and therefore not a new implementation of the uncopywriteable algorithms expressed in Fruit but a derivative work in the copyright sense. Lexmark International, Inc. v. Static Control Components, Inc. accepts that conversion of a function to lookup table, or rearranging terms in an expression are not by themselves original acts. However, the transformations in Rybka go well beyond mechanical changes at the call graph level.

They have instead gone up to a higher level of abstraction shown that the code in Fruit, with different input parameters than the Fruit defaults, can generate numbers which after post-processing match numbers used in Rybka. They have stated that the order of certain actions, where the order should be arbitrary, is consistent between the two programs.

While this was enough to convince the judges that Rybka contained unacknowledged algorithmic influence from Fruit in the fashion required by the rules, this argument is not sufficient to show copyright infringement. Here too the comparison method should be validated by applying it to other programs which use the same algorithmic approach as Fruit and which are known to not have a shared copyright history. The Rybka investigators have failed to do this.

Functional similarity by itself does not show copyright infringement. The clean room design model in programming is a well-understood method of using reverse engineering to create a new implementation which does not infringe on copyrights or trade secrets. One group of people reverse engineer the software and develop a design document which is reviewed by a lawyer to make sure that no copyrighted material is included. The document is passed to another group of people to implement, where the people in the second group were selected because they have no knowledge of the original implementation. An real-life example is OpenOffice, which is a clean-room implementation of Microsoft Word and is close enough in functionality that people can switch from one program to the other with little trouble.

Rybka was not developed as a clean-room reimplementation of Fruit, and the author has long acknowledged an intensive review of the Fruit source. The point here is only to show that functional similarity is not a clear indicator of copyright infringement.

Some may believe that reviewing the source code to Fruit means that Rybka is necessarily a derivative work. This is incorrect. A clean-room reimplementation makes it abundantly clear that there is no copyright infringement, but the lack of a clean-room style does not mean there is infringement.

One of the most widely used computer operating systems is Linux, which was greatly influenced by the MINIX operating system. MINIX was available in source form under a proprietary license, and there was a college textbook which described how it worked and included most of the source code. The Linux author studied MINIX and ran MINIX on his computer before developing Linux. The MINIX author stated clearly that Linux is not a derived work in the copyright sense, and no review of the Linux source code has ever shown copyright infringement to any other code, including MINUX.

Therefore, intensive review and study of the Fruit does not mean that Rybka infringes on the Fruit copyright just like an intensive review and study of MINIX does not mean that Linux infringes on the MINIX (or Unix) copyrights.

The Rybka investigators acknowledge that no individual piece of evidence is the proverbial smoking gun. Instead, the pattern of pieces reveals the infringement. The history of evidence synthesis like this is fraught with methodological problems. Quoting from http://en.wikipedia.org/wiki/Meta-analysis#Disadvantages_and_weaknesses:

> The most severe weakness and abuse of meta-analysis often occurs when the person or persons doing the meta-analysis have an economic, social, or political agenda such as the passage or defeat of legislation. Those persons with these types of agenda have a high likelihood to abuse meta-analysis due to personal bias. For example, researchers favorable to the author's agenda are likely to have their studies "cherry picked" while those not favorable will be ignored or labeled as "not credible". In addition, the favored authors may themselves be biased or paid to produce results that support their overall political, social, or economic goals in ways such as selecting small favorable data sets and not incorporating larger unfavorable data sets.

There are ways to help offset these problems. For example, all comparison methods should be reported before doing the analysis, along

with the definition of what "infringing" means for that case. Methods which fail to report similarity must be recorded. All participants must state possible sources of bias, and the method for selecting the participants must also be published.

This was not done. Of course, if there was strong evidence for copyright infringement then a careful synthesis of the evidence would not be needed, but that was not the case here. Instead, the results seem very much cherry-picked.

It may very well be that Rybka contains copyright infringing code. It's possible, after all, that 27 lines out of 525,000 are enough to make a substantial copyright infringement claim. The problem is that the methodology of the Rybka investigators is not strong enough to be convincing.

They claim to use the abstraction-filtration-comparison test to determine substantial similarity, but without the appropriate filtration. At each of the structural levels they fail to show that the discovery methods are not producing false positives, and they fail to demonstrate that the similarity level is greater than would be expected from a non-infringing chess program implementing the idea at the same structural level.

The similarity between Fruit and Rybka is strongest at the highest level of the analysis, but the abstraction-filtration-comparison test acknowledges that at a high enough level there's no copyright protection. This is due to the merger doctrine.

The document http://www.top-5000.nl/ZW_Rybka_Fruit.pdf is highly misleading. Without a mapping from the C structures based to disassembled code, it's impossible to judge the correctness of what the author writes. With a translation directly into idiomatic Fruit code, the visual similarities are overly distracting from the actual comparison, which is on a level that isn't show.

(You saw in my first email that just because someone reports what the code says, it doesn't mean that that's what the code actually does.)

Copyright law already acknowledges that at higher levels there's no copyright infringement because it's different expressions of a common idea. Hence the statement "high-level functionality is always equivalent in these cases" is meaningless unless it's established that this level is not high enough.

Cheers,

Andrew

This is the start of Zach Wegner's document as used as a base of the ICGA-Rybka investigation commented, analysed and is the collected work of several chess programmers although expressed in my own words.

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.

Not if one looks well enough, we counted over 20 differences in EVAL on minor but also on fundamental and dominating EVAL ingredients of a chess program. Basically nothing is the same especially not the dominant factors in EVAL such as Mobility, King Safety and Passed Pawns, see for instance a study about the ELO values of an evaluation function.

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.

While it's our opinion the Rybka investigators seriously tried to list the *general differences* (as the header states) the below list is extremely poor and yet accusing at the same time. Since we have spot over **36 of such *general differences*** we must conclude the Rybka investigators missed a lot. We will discuss as they come along in this document.

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.

This statement is highly suggestive and assumes things that can not be proven, more the *since the end result is the same* simply is not true as we will demonstrate.

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.

This is false. Nothing of Fruit's special endgame knowledge is in Rybka, not in the code not in Rybka's material imbalance table.

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.

We assume with *unnatural* Wegner means the value of a pawn as used in Rybka 1.0 which is 3200 contrary to Fruit which uses the natural value of 100. However what is labelled as *unnatural* in fact is a brand new idea in computer chess now practiced by almost everybody. The goal of 3200/32 and the end of Rybka 1.0 EVAL is to have more equal scores but better rounded -> more beta-cutoffs -> (somewhat) faster search.

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.

To understand what's been said here: it is assumed Rajlich took the Fruit PST initialization code (pst.cpp lines 86-288) changed the 17 parameters weights (pst.cpp line 32-48) added an 18th parameter plus own weight and then imported the output into Rybka.

Meaning that contrary to Fruit there is no PST code in Rybka.

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 };
```

Note that none of these tables are in the Rybka binary as explained above.

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	Rybka

`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;`
	`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;`

Note that there is no such code (right column) in the Rybka 1.0 binary as explained above. It's back-reasoned non-existing code based on the assumption Rajlich used Fruit's initialization code to create his own PST's.

The same is true for the below listed imaginairy Rybka code for Knights, Bishops, Rooks, Queens and Kings. It's not in the Rybka 1.0 binary. Therefore we leave that without comments, all the yellow code is not in the Rybka 1.0 binary.

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

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,square_opp(sq),Opening) += 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;`
 	
`...`
`for (sq = 0; sq < 64; sq++) {`
`  P(piece,sq,Opening) +=`
`RookFile[square_file(sq)] *`
 	
`  RookFileOpening;`
`}`
	`static const int RookFileOpening = 104;`
	
	`...`
	`for (sq = 0; sq < 64; sq++) {`
	`  P(piece,sq,Opening) +=`
	`RookFile[square_file(sq)] *`
	
	`  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 QueenCentreEndgame = 4;`
`static const int QueenBackRankOpening = 5;`
 	
`...`
`for (sq = 0; sq < 64; sq++) {`
`  P(piece,sq,Opening) +=`
`QueenLine[square_file(sq)] *`
`  QueenCentreOpening;`
`  P(piece,sq,Opening) +=`
`QueenLine[square_rank(sq)] *`
`  QueenCentreOpening;`
`  P(piece,sq,Endgame) +=`
`QueenLine[square_file(sq)] *`
`  QueenCentreEndgame;`
`  P(piece,sq,Endgame) +=`
	`static const int QueenCentreOpening = 98;`
	`static const int QueenCentreEndgame = 108;`
	`static const int QueenBackRankOpening = 201;`
	
	`...`
	`for (sq = 0; sq < 64; sq++) {`
	`  P(piece,sq,Opening) +=`
	`QueenLine[square_file(sq)] *`
	`  QueenCentreOpening;`
	`  P(piece,sq,Opening) +=`
	`QueenLine[square_rank(sq)] *`
	`  QueenCentreOpening;`
	`  P(piece,sq,Endgame) +=`
	`QueenLine[square_file(sq)] *`
	`  QueenCentreEndgame;`
	`  P(piece,sq,Endgame) +=`

```
QueenLine[square_rank(sq)] *
 QueenCentreEndgame;
}
for (sq = A1; sq <= H1; sq++) {

 P(piece,sq,Opening) -= QueenBackRankOpening;
}
```

```
QueenLine[square_rank(sq)] *
 QueenCentreEndgame;
}
for (sq = A1; sq <= H1; sq++) {

 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;`

Piece Square Tables

```
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)] *
```

```
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;
}
```

```
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.

While we agree that the fiddling with the Fruit code and its parameters gave an impressive result we think the conclusion is premature. When looking at programs (such as Fruit and Rybka) that generate PST's by calculating, *contrary* to hand typed (!) PST's, we see there is so ***little information*** in the piece_square cells and thus that random equalness is more the norm than the exception when you are actively start looking for them by fiddling with the parameters.

We think that several chess programmers but especially programmer Miguel A. Ballicora succesfully has demonstrated this phenomenon by his various experiments finding exact PST or semantical equivalent PST's in various other chess programs. [1] [2] [3]

Rajlich when we asked him for the utility that created the Rybka 1.0 PST's

1. **Vasik Rajlich**: The piece-square table C# code - unfortunately I have only the code which creates my piece-square tables today. The piece-square tables are similar but not exactly the same as the Rybka 1 piece-square tables. Also, I definitely have tinkered with the C# code in the last six years. For example now I use .NET reflection, which AFAIK was not even around in 2005. So, it won't be exact. Plus, I'll probably want to delete a few things. Is this really worth doing? It's hard for me to see this as a major issue. (December, 2011)

2. **Vasik Rajlich**: During my tuning I used ints, but much finer than pawn=3200. I needed to be able to "perturb" each eval weight minimally and calculate the delta for the fit between eval scores and game results. This is the "gradient" part of gradient descent. (December 2011)

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;
     }
```

The Fruit QUAD function (unique Fabien DNA, noone has except clones) is absent in Rybka 1.0 In fact the lack off is evidence Rajlich wrote his own code. The fact Wegner fails to mention this is troubling.

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	Simplified version

`min = PassedEndgameMin;`
`max = PassedEndgameMax;`
`delta = max - min;`
 	
`...`
`// misc. bonuses`
`delta += bonus;`
 	
`...`
`eg[att] += min;`
`if (delta > 0) eg[att] += quad(0,delta,rank);`
	`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:

Nothing we have seen so far is unusual in a chess program. The Rybka=Fruit message is uncalled for.

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.

We don't understand the Rybka=Fruit link made here, what is described is standard chess knowledge in every chess program that a) wants to survive in a pawn ending or b) not to wrongly enter into a lost pawn ending. Such code including pawn races including promoting with checks has been in Rebel since the 80's.

We fail to see how general known concepts of chess knowledge can be used as evidence for code theft. Such hints to guilt are misleading, even more misleading is the phrase *and they both receive the same bonus* as what other bonus should an unstoppable pawn be given other than Queen value - Pawn value ?

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).

Again, we fail to see the relevance. Checking squares before a passed pawn is the most natural thing to do in a chess program, just commonly used chess knowledge in use by everyone. Fruit evaluates only one square in front, Rebel 3 squares, Rybka 1.0 can do it till the promotion square because of the bit-board structure. The latter obviously should work to Rajlich's advantage and not the other way around as can be read between the lines.

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

```
if (board->piece_size[def] <= 1
&& (unstoppable_passer(board,sq,att)
|| king_passer(board,sq,att))) {
delta += UnstoppablePasser;


} else if (free_passer(board,sq,att))
{ delta += FreePasser;
}
```

Rybka

```
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)


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.

Nothing new under the sun here also, any good chess program has it.

No evidence of any wrongdoing other than being suggestive.

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

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.

This is one of the worst examples in Wegner's document, another example how to make Rybka 1.0 look like Fruit while the material imbalance table was a brand new idea in computer chess nowadays in use by many.

First of all Fruit has NOT a material table such as Rybka 1.0, Fruit has a hash-table that is maintained during search whereas Rybka has a fixed large pre-computed material imbalance table inside the executable with a size of over 1MB.

Rybka has almost all things different compared to Fruit but in the Wegner's document it is almost identical.

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; };
```

Attention to what's said here. The structure*isn't certain*, it *seems likely* that, the structure*is something like*..... No wonder all the wrong conclusions.

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.

From Fruit 2.1, material.h:

```
const int DrawNodeFlag = 1 << 0;
const int DrawBishopFlag = 1 << 1;
const int MatRookPawnFlag = 1 << 0;
const int MatBishopFlag = 1 << 1;
const int MatKnightFlag = 1 << 2;
const int MatKingFlag = 1 << 3; < sab="13721" span>
```

Meaning, Fruit has six flags. Four of them (as Wegner admits) are not in Rybka. But then Wegner continues Rybka has **all** other flags that are in Fruit. 6-4=2 means **all** Wegner? And further down we read looking at the left-to-right comparison that the "MatKingFlag" implementation is similar (but different), and that the "DrawBishopFlag" implementation is (said to be) the same but its usage is completely different in Rybka.

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.

As already stated everything is different as the code itself shows. We don't understand the relevance if your goal is to proof code theft, this more or less is evidence of the opposite, original ideas and own code.

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.

Again, we fail to see the relevance. Fruit has no Lazy Evaluation. It's just another sign of Rybka originality. Assuming Wegner is right about the Lazy Eval boolean flag in the Material Imbalance Table (remember he is not sure) then we are witnessing another brand new idea in computer chess.

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.

The Fruit interpolation code was not new as Wegner suggests, it was already practiced by Phalanx in 2000 5 years before Fruit 2.1.Phalanx XXII is open source, check out the variables "midresult" and "endresult".

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;`

Why needed when already pre-calculated in the MIT ?

```
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.

Not only misleading but likely false as well. The above is a very powerful picture that suggest verbatim copying and yet no such code is in Rybka 1.0 at all. The reader is to believed *Rybka has the same formula as Fruit* without mentioning the absence of code in the Rybka 1.0 binary.

And further we have Wegner's public statement: *Rybka has a different phase calculation* saying the exact opposite. And now we are really confused.

The immediate question that springs to mind is that if "phase" is pre-calculated in the MIT (as Wegner stated but isn't entirely sure) then why is there a need to calculate it?

Furthermore there is no mentioning Wegner has checked all the values (25 combinations) of "phase" in the MIT to match the Fruit code and also there is the absemce of some form of evidence he checked all the combinations.

A quote from the below text: *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.*

So Wegner is blaming Rajlich not to have understood the Fruit code he allegedly copied while on the other hand Wegner isn't even sure about the structure of the MIT following his own words:

Wegner - *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:*

And considering all the other mistakes Wegner made with the MIT it's not Rajlich who did not understood but Wegner not understanding Rybka's MIT.

And related, it must be said, there is no single word spoken during the Panel deliberations when Rybka was investigated about mistakes like these and others already addressed. The Panel discussions contained only 214 postings of which 40% polling for a readyness of the members to vote followed by the voting itself. At best that leaves 140 postings regarding research and debate all the technical and complicated issues.

This in sharp contrast after the "guilty" news broke and the fora exploded. There are over 15,000 posts alone in the Rybka forum, the real investigation happened after the Panel deliberations and this document is a reflection what has been discussed.

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.

On September 8, 2012 Zach Wegner replied to our question about the non-existing "phase" code in Rybka. When asked to quote his answer (bald or rephrased) he would not allow us.

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.

Indeed, let's say a few words about mail-box (Fruit) and Rybka 1.0 (bit-board) since the topic isn't addressed yet. These are 2 total different data structutes that guarantee one thing: total different code. As such it is impossible to proof code theft for EVAL.

To make a case for code theft after all Wegner resorts to Fruit - Rybka EVAL similarities that are present in every chess program, common (non-copyrightable) chess knowledge found in every instructive chess book. We fail to see the relevance to code theft. And when we scrutinize the below given evidence we can not even agree with Wegner's suggestive: *Rybka's pawn evaluation is very simple. It is again, virtually identical to Fruit's* because Rybka's implemented chess knowledge about Pawns is different than Fruit's.

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-bitfile-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.

Pawn Hasing is a common technique in chess programs that pre-dates Fruit and Rybka by several decades and so we don't see the relevance to mention it. As one can see Rybka 1.0 has it coded different than Fruit and is another sign Rajlich wrote his own code.

While Wegner's document is suggestive and preconceived of guilt he is careful enough not to speak what he really thinks (code theft) in this document but in several public statements he is more open, a few examples:

1. *I came to the conclusion, after seeing what I saw, **that Rybka started its life as Fruit***. [more]

2. *It's very clear to me that the reason Rybka 1.0 was strong was **that it took Fruit 2.1**, tuned the parameters, made some other inconsequential changes, and sped it up quite a bit. Why anyone would want to consider Rybka an original engine in light of the pages I posted is beyond me.* [more]

3. *Coming back to R1/Fruit, yes, if you look at each example in detail, you can't say there is much hard evidence of direct code copying. As I said to Vas, I'm only completely certain that three characters were copied ("0.0"). But given the entire picture (how similarities to Fruit are all over the place, and the previous Rybka versions shared no code) **it's just so obvious to me that Vas took Fruit as a base** and rewrote things on top of it, presumably until he felt it was "clean".* [more]

   Personally we prefer this kind of openness ourselves as it reflects the truth and heart of the matter, this case was never about ICGA rule #2 but to punish Rajlich for copying Fruit 2.1 under less strict rules (originality) than copyright infringement.

   Or by the words of Letouzey author of Fruit:

   Hi Ed,

   Yes, this is my conjecture from looking at the Strelka source code which as confirmed by experts is a faithful C translation of the Rybka binary:

1) Vasik took the Fruit 2.1 public source code
2) he removed everything that did not hurt Elo rating and could gain some speed
3) he moved all possible code into static tables or hashable stuff (e.g. pawn shield/storms were moved to hashed pawn eval)
4) he converted the board to bitboards, probably by maintaining the two data structures until he finished the conversion
5) he (probably) manually inlined some other stuff

Examples of 2) are collecting the PV during search, mate-distance pruning and underpromotions (this makes move generation faster by avoiding having to test for promotion at all).

The rationale of this process is as follows:
a) speed is the priority; Fruit is slow and simply making it faster is a simple way to get to the top of rating lists
b) simplify the code as much as possible (move to tables, inline), this will help compiler optimisation too; Fruit was designed with future development in mind so it had many unnecessary function calls
c) obfuscation was less of a priority; usually a) and b) automatically bring c)
d) it's possible to test the engine at anytime to make sure no big mistake was made

About c), it seems Vasik cared about hiding copied Fruit code like UCI parsing only after Strelka appeared. He considered a binary was safe enough from investigation.

So when the TRUE reason for the invetigation is code theft then let's talk code theft (Rajlich starting from the Fruit 2.1 source code) instead of listing commonly used non-copyrightable chess knowledge that frequently by Wegner is labeled as *virtually identical to Fruit's* and then when looking at it is not. **In a later stage we will publish our own findings what *possibly* (emphasis added) could have been borrowed and if that falls under copyright infringement.**

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.

Let's only respond to the yellow not to fall into endless repetitions. There is a serious logic flaw in the above. If one suggest that the order of coded elements in EVAL is evidence for code theft then the same principle should apply for the whole EVAL and all its elements. And the truth is that the order of EVAL in FRUIT is not equal nor similar than the order of EVAL in RYBKA.

Sometimes the best counter evidence is the evidence that unburdens a suspect, the DNA traces of copying that should have been there but where not. Besides when looking at the binary the semantic structure between Rybka and Fruit is quite different. In essence: Rybka evaluates directly, not afterwards as Fruit does, also no variable maintenance as in Fruit, no need doing it the Rybka way.

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 (MaskPawnDoubled[square] & Board.pieces[WP])`

No relevance again, there is no copyright on double pawns and every chess engine has that inside. In

stead the difference in implementation is more interesting, Rybka has no penalty for middle-game double pawns, Fruit has. The hand and preference of an IM.

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;`
	`  }`
	`}`

Not much to add here, the concept of penalizing an isolated pawn on an open file is standard chess knowledge found in every instructive chess book. It's in Rebel since the 80's and surely in any other decent chess program.

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

```
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;
```

Rybka

```
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;
}

}
```

```
}
}
```

We beg to differ. Fruit's backward pawn is different, quite unique we must say, it has code for second ranked pawns that can advance 2 squares. Rybka has not, just a quick and dirty evaluation. Which program before Fruit had this feature? And why since the accusation is that Rybka 1.0 started her life as Fruit 2.1 did Rajlich removed that?

It's pretty clear by now that Wegner's opening statement on Pawn evaluation, we quote from above:

Rybka's pawn evaluation is very simple. It is again, virtually identical to Fruit's.

is an exaggeration.

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];`

Indeed as Wegner states, (already) discussed elsewhere.

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

```
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] +=
```

Rybka

```
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];
}
}
```

```
quad(CandidateOpeningMin,CandidateOpeningMax,rank);
endgame[me] +=
quad(CandidateEndgameMin,CandidateEndgameMax,rank);

}
```

While we have not researched this specific piece of chess knowlledge the first thing that springs in mind are not the similarities Wegner speaks about but the above left-right code visualization. If one wants to make a case for code theft this isn't a very good example.

Of course the issue is addressed by Wegner, Rybka is bit-board. But so is the life of a reverse engineer programmer who wants to proof mail-bord = bit-board. It can not be turned against Rajlich, in fact it is Rajlich strongest point of originality, the bit-board data structure that guarantees total different code as this example so convincly demonstrates.

As a last item again Fruit's QUAD function. Not in Rybka.

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.

Mobility is one of the most debated issues regarding the Fruit / Rybka controversy.

Wegner's opening statement again is accusing, the reader is told what to think and Rajlich strongest point (a bit-board engine) is undermined on beforehand and more or less works against Rajlich than

in his favor. Bluntly said, If there is a difference between Fruit and Rybka blame it on the bit-boards and the difference magically disappears.

We will proof Wegner wrong but first let's first read further.

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;
```

During the debate the best analysis was given by programmer Sven Schuele [whole post in context]

I am on no "bandwaggon", Bob. Please avoid such wording, it is inappropriate since you are trying to downplay my contribution as if I were a newcomer involved for the first time. You know my position in the RF issue pretty well, for years.

I think that "Cray Blitz" is not very relevant in our discussion since it has the same author as Crafty, so I suggest that you avoid referring to CB. The points you raised about CB vs. Crafty were never put under question AFAIK but don't contribute anything here.

Your notion of "calculating attacks on the fly" is different from mine. Either you look up information in a table, or you calculate that information "on the fly", you don't do both. The rotated bitboard approach as well as "magic bitboards" both include a "lookup" concept for sliding attacks, albeit in a different way. No need to explain that further **to you, Bob.** This is in contrast to other concepts like "dumb7fill", for instance, where you really have "on the fly" calculation of attacks.

Having clarified the wording, I think we have the following situation (here is also the promised part of my reply to the parallel sub-thread where you replied directly to Ed):

Fruit 2.1:
- Feature definition includes not to count attacks to friendly pieces.
- Implementation is "mailboxy" using loops to calculate attacks "on the fly" and count mobility. This is quite expensive but there is probably nothing substantially faster for a mailbox program.
- Scoring is linear.
- Everything follows in a straightforward manner from "feature definition + board representation + scoring strategy".

Crafty up to 20.1:
- Feature definition includes to also count attacks to friendly pieces.

- Implementation uses table lookup in a way typical for rotated bitboards. The precalculated information being looked up is the mobility count itself, which is only possible due to not excluding friendly pieces.
- Scoring is linear.
- The implementation is quite efficient within the conditions given by feature definition and board representation. It is also "straightforward", since not doing a table lookup with rotated bitboards would make absolutely no sense.

Rybka 1.0beta:
- Feature definition includes not to count attacks to friendly pieces.
- Implementation uses table lookup in a way typical for rotated bitboards. The precalculated information being looked up is only the attack set.
- Scoring is linear.
- It seems to be most efficient under the given conditions which do not allow to store the mobility count in the lookup table. It is also "straightforward" for the same reason as in case of Crafty above.

Crafty 20.2 and later, pre-magic:
- Same as older Crafty above, with the exception of introducing non-linear scoring.

Crafty after switching to magic bitboards:
- Feature definition still includes to also count attacks to friendly pieces.
- Implementation uses database lookup typical for magic bitboards. The precalculated information being looked up is still the mobility count itself, which is only possible due to not excluding friendly pieces.
- Scoring is non-linear.
- The implementation is quite efficient within the conditions given by feature definition and board representation. It is also "straightforward", since not doing a database lookup with magic bitboards would make absolutely no sense.

So the conclusion is obvious: in each of the cases above we see an implementation that is "typical" and follows in a "straightforward" manner from design decisions about
- the exact definition of the "mobility" feature to be applied,
- the given board representation (and the method of calculating sliding piece attacks which is related to it), - and the scoring strategy for the feature. All implementations are very different from each other, where of course the pre-magic Crafty versions are the closest ones. All implementations, perhaps with the exception of Fruit, are also heavily driven by efficiency issues, and appear to be most efficient under the specific conditions.

```
Therefore I see no reason to assume any "derived code" relation between the engines
mentioned above regarding mobility evaluation, except "Crafty vs. Crafty" of course
which we don't need to discuss. All that is possibly shared between Crafty, Fruit, and
Rybka is clearly on the abstract design level. But in none of the relevant cases there
is even a sharing of the complete design, as shown above.
```

We can also turn the complicated to the simple and say it in a few words. Common sense states that when you write code several times faster, pack 4 expensive loops (see eval.cpp lines 621-642) into 1 instruction `[ mob = popcnt(attacks & ~own_pieces); ]` you are the better engineer. Furthermore there is no copyright on chess knowledge, no copyright on commonly used concepts such as mobility especially not when it's explained in detail on the Wiki chess pages and plenty of other sources on the Internet.

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;
 }
 ...
}
```

A quiz question. What's wrong with the reversed engineered Rybka code ?

It's:

```
opening += RookOpenFileOpening - RookSemiOpenFileOpening;
endgame += RookOpenFileEndgame - RookSemiOpenFileEndgame;
```

Note that the compiler would merge "RookOpenFileOpening - RookSemiOpenFileOpening" into one value since both are declared as a CONSTANT. So where are these 2 constants (1035 and 428) coming from? They are not in the RYBKA executable, only the values 971 and 172 are. As confirmed by the assembler:

RYBKA original ASM code

```
401b50: 48 85 05 39 ca 26 00 test QWORD PTR [rip+0x26ca39],rax
401b57: 75 3d                jne  0x401b96
401b59: 83 c6 40             add  esi,0x40  // 64
401b5c: 81 c7 00 01 00 00    add  edi,0x100 // 256
401b62: 4c 85 f8             test rax,r15
401b65: 75 0c                jne  0x401b73
401b67: 81 c6 cb 03 00 00    add  esi,0x3cb // 971 (1035-64)
401b6d: 81 c7 ac 00 00 00    add  edi,0xac  // 172 (428-256)
```

And thus Wegner in order to proof FRUIT code ASSUMED Constant Folding taking place as part of the Optimizing Process of the compiler ?

In science, unfolding a constant "A" into "B - C" just because you assume it is justified, **without any explanation**, is literally, academic dishonesty. In science you are supposed to present the data as it is, transparently.

Unfortunately we are forced to stop here commenting Wegners document due to all the scrambling of the fonts, fontsizes and tables that took place during the conversion from PDF to HTML. Basically the HTML is hardly editable and the formatting takes more time than adding the comments.

Nevertheless we think we have demonstrated that there is no single sign of code theft.

Before we move to the more interesting document of Mark Watkins (as it does hint to code copying) we like to address one more essential difference between the Fruit 2.1 and Rybka 1.0 EVAL namely that Fruit evaluates in 2 steps whereas Rykba directly adds

to score. We don't understand why Wegner omits such a fundamental difference. One example of the many is found in Fruit's EVAL.CPP

FRUIT: EVAL.CPP -> LINE 655-682

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

And later that op[me] and eg[me] are updated to the REAL evaluation variables: openingand endgame

FRUIT: EVAL.CPP -> LINE 759-762 // update

```
*opening += ((op[White] - op[Black]) * PieceActivityWeight) / 256;
*endgame += ((eg[White] - eg[Black]) * PieceActivityWeight) / 256;
```

RYBKA just having:

```
opening += RookSemiOpenFileOpening;
endgame += RookSemiOpenFileEndgame;
```

Directly adding to the score.

Chapter II
Mark Watkins document

Watkins on various places in his document hints to copy theft, we like to address these points.

1. setjmp (idea borrowed from Fruit by the admission of Rajlich to unwind the recursive search) (elo gain 0) also found in other programs, for instance in Rookie by Marcel van Kervinck.

2. Root search time control similarities playing apparent obvious moves (recaptures etc.) faster. (elo gain 1-2)

3. Calling search from UCI (almost identical code) (elo gain 0) It's place is not so unusual as suggested, a version of

Glaurung does it too.

We don't consider these first 3 points as a sign of wrongdoing but your mileage may vary of course.

4. More serious is Watkins accusation in Chapter 6.3.1 Parsing the "position" string. Please find our analysis here.

5. We also researched Watkins accusation in Chapter 6.3.2 Time management. Please find our analysis here.
   It was disappointing for us to learn that the error with "0.0" was KNOWN by Watkins one year before the ICGA investigation started (2010) but was withheld from his document, withheld from the Panel and withheld from every forum discussion long after the ICGA verdict. For many programmers (Wegner included) "0.0" was an hot issue confirmed by the hundreds postings on this issue alone.

We believe we convincingly refuted [4] and [5]. As such we believe the real evidence for code theft is extremely poor.

As a final remark on this chapter we like to add it's not unlikely to *assume* Rajlich modelled Rybka 1.0 EVAL to Fruit. There are signs for that and much of the contrary as demonstrated in this document. *Probably* the truth is somewhere in the middle. The latter more or less confirmed by Rajlich himself, 3 weeks after the Rybka 1.0 release (December 2005) in an interview his own words were: I took many ideas from Fruit. Also in the documentation Fruit is mentioned as an inspiration source.

Special Thanks

I hesitate to include this section because I know I'll forget people who have been helpful in this project, but (with advance apologies to the omitted) here goes:

Robert Hyatt - For Crafty. There is nothing like an open source program for passing knowledge to the next generation.

Fabien Letouzey - For Fruit, which shattered a number of computer chess myths, demonstrated several interesting ideas, and made even the densest of us aware of fail-low pruning.

TordRomstad - For making Fabien aware of fail-low pruning :-), and more seriously for sharing in every way possible his considerable knowledge.

Eugene Nalimov - For his cryptic but somehow fully functional endgame tablebase access code.

Uri Blass, Gerd Isenberg, Dieter Burssner, Vincent Diepeveen, Raschid Chan, Anthony Cozzie, Mridul M :), Thomas Gaksch, Peter Berger, SandroNecchi, Ed Shroeder, Amir Ban, Christophe Theron and every one else, past and present, on the computer chess club: For sharing their computer chess knowledge despite the fact that in principle computer chess is a competitive field.*

Heinz van Kempen, Guenther Simon, Olivier Deville, Sergio Martinez, Claude Dubois: for testing early versions of Rybka despite countless bugs and annoying problems.

Alex Dumov, Gabriel Luca: for helping a Windows newbie get up to around half-speed without excessive derision (or at least open derision :))

and Iweta: for being great! :) and a pretty good Rybka tester and web master to boot

Happy testing, and best chess regards,

VasikRajlich

Budapest Hungary

December 4, 2005

Chapter III
External evidence

Contrary to the ICGA press release which stated that Rajlich did not cooperate with the investigation Rajlich **did** offer cooperation and defence, see the email correspondence and offered the "**Ponder-hit**" defence.

However Rajlich's defence was ignored and overruled by David Levy and Rajlich's defence never reached the Panel who was in charge of researching all evidence as stipulated by the Charter David Levy notable wrote himself.

"Ponder-hits" and its twin "similarity tester" are 2 statistical tools that measure the move similarity between engines and especially the results of "similarity tester" are surprisingly because it produces a sorted list of all the known clones.

And Rybka 1.0 is not one of them which confirmed our findings.

The evaluation originality test

Originality testing using Don Dailey's Similary tester utility
60% and higher being a sign of non-originality,
65% and higher being a sign of cloning.

Similary tester is an utility that is written by Komodo programmer Don Daily that measures the similarity between chess programs. Each program has to analyze 8000 chess positions where multiple good moves are present and doing so basically a profile of the evaluation (playing style) of a program is made. An example:

It's widely known that Loop and Onno are Fruit clones. Similary Tester reports as follows:

Fruit 2.1	-----	70.81%	66.75%
Loop 2007	70.81%	-----	68.56%
Onno 1.04	66.75%	68.56%	-----

It's also widely known Houdini is derived from the Ippolit family. The statistic:

Houdini 1.0	-----	67.27%	71.07%
Ippolit 0.080a	67.27%	-----	69.31%
RobboLito 0.09	71.07%	69.31%	-----

And now for Rybka, Fruit and Doch 09.980

Doch 9.980	-----	51.58%	50.30%
Fruit 2.1	51.58%	-----	54.43%
Rybka 1.0 Beta	50.30%	54.43%	-----

Doch 09.980 is Don Dailey's return to the computer chess arena in 2009 after a long absence, an engine (like Rybka) coming out of nothing with an incredible CCRL rating of 2970 elo. Both Doch and Rybka seem to have no connection to Fruit, they both are relative safe and far away from the 60% gray area and only diifer a neglectful 3 percent in comparison with Fruit 2.1

Furthermore both authors don't hide the origin of their success: the open-sources. From the Doch 09.980 README file we read:

Also, much credit goes to the authors of open source chess programs.

Many of the ideas and techniques for doch have been borrowed from these wonderful works of art.

If Doch 09.980 is an original engine then so is Rybka 1.0 Beta.

We want to emphasize that the last thing we want is to question is the integrity of programmer Don Dailey, we don't. The above information is strictly meant as another piece of counter evidence to unmask the ICGA verdict as a regrettable moment in the history of computer chess.

Somehow (we believe) it's (still) not clear that the post-internet generation learns 10 times faster than the pre-internet generation because of the existence of email, open sources and computer chess fora which caused an explosion of knowledge and that therefore quick progress can be established without copying, as proven by many authors: Fabien Letouzey in the first place, Vasik Rajlich, Anthony Cozzie (Zappa) and lately Don Dailey (Doch, Komodo).

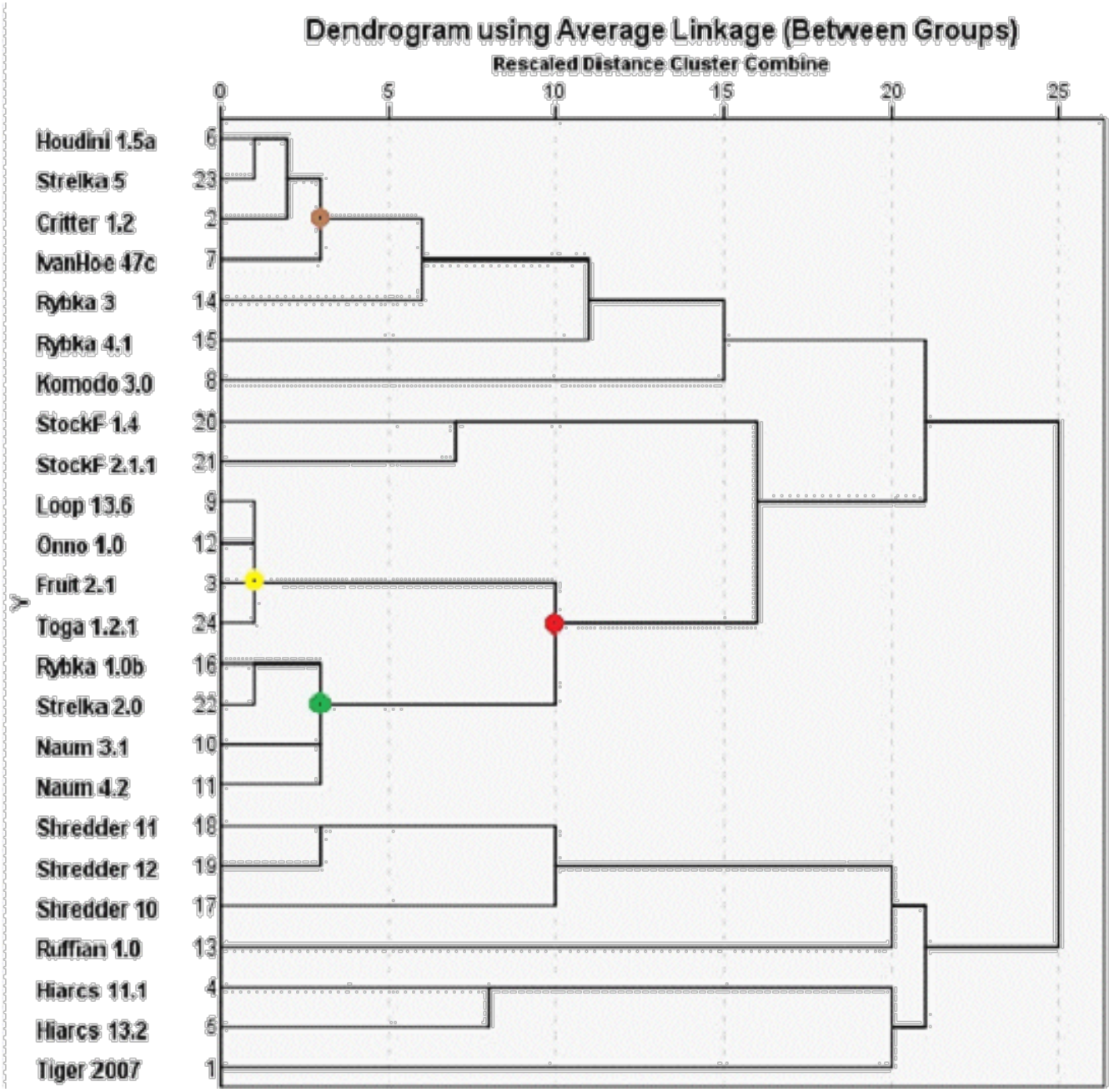
Like Rajlich also Letouzey came out of nothing. Fruit 1.0 dates from March 2004, Fruit 2.1 is 15 months later and it took quite some known names by surprise.

For more information about Don Dailey's Similary tester utility check out the study of Adam Hair on the issue how to detect clones and how well the software creates a sorted list of suspect engines.

The Ponder-hit system
The Rajlich defence

Another way to test the originality of a chess program is the "ponder-hit" system as introduced by Kai Laskos, a computer chess theorist with statistical expertise who posts in the Talkchess computer chess forum.

From a large volume of CCRL games the "ponder moves" were extracted from the PGN and collected in a similar way as Don Dailey's Similary tester. And from the data (like in the Adam Hair study) a dendrogram was created.



The picture tells the same story as Don Dailey's Similary tester, the same usual suspects and Rybka is not one of them. Loop, Toga and Onno being Fruit clones, Strelka and likely also Naum being Rybka clones.

The TOP-30 of Adam's study

Some additional notable pairs of engines (from the entire sample of 378 tested engines) not noted to be from the same family, with the exception of acknowledged Fruit/Toga derivatives and Strelka (their inclusion is for illustrative purposes):

```
 1. Fruit 2.2.1         vs  Loop 2007          84.73% (similarity)
 2. Fruit 2.2.1         vs  Loop 10.32f        84.13%
 3. Strelka 1.8         vs  Rybka 1.0 Beta     73.11%
 4. Fruit 2.1           vs  TogaII 1.0         72.43%
 5. Fruit 2.1           vs  Loop 2007          70.81%
 6. RobboLito 0.085d1   vs  Houdini 1.00       70.37%
    ................
29. Glaurung 1.2.1      vs  Alfil 8.1.1        59.47%
30. Rybka 3             vs  Houdini 1.00       59.37%
```

Note that Rybka 1.0 similarity to Fruit 2.1 is only 54.43% and is far away of the red zone.

Chapter IV
Historic evidence

History of cloning

1989 - Quickstep - Richard Lang clone (copyright breach)
1996 - Gunda - Crafty clone (copyright breach)
2002 - El Chinito - Crafty clone (copyright breach) Author admitted the wrong doing.
2003 - List - I leave this case out. Reul was banned but later reinstated.
2006 - Lion++ - Fruit clone (copyright breach) Authors admitted because felt they did nothing wrong.
2010 - SquarknII - Robbolito version with only 3 instructions changed.

All good decisions. All have in common that (source) code was taken as a whole and a derivative was the result. In none of the cases the IGCA rule #2 and its interpretation became under pressure. Copying was proven, end of story. A second ingredient what they have in common *is that all of them were easy to debunk as a clone.*

The pattern (easy catch) continues for engines that did not take part in ICGA tornaments, a known example is:

2010 - Houdini 1.0 - Derived from Robbolito. It took me one hour to spot that from the 2 binaries.

Robolito 0085e	Houdini 1.0

```
0044903C aKingWhite1     db 'king white != 1',0
0044904C aKingBlack1     db 'king black != 1',0
0044905C aQueenWhite9    db 'queen white > 9',0
0044906C aQueenBlack9    db 'queen black > 9',0
0044907C aRookWhite10    db 'rook white > 10',0
0044908C aRookBlack10    db 'rook black > 10',0
0044909C aChiaroWhite9   db 'chiaro white > 9',0
004490B0 aChiaroBlack9   db 'chiaro black > 9',0
004490C4 aDarkWhite9     db 'dark white > 9',0
004490D4 aDarkBlack9     db 'dark black > 9',0
004490E4 aBishopWhite10  db 'bishop white >
10',0
004490F8 aBishopBlack10  db 'bishop black >
10',0
0044910C aKnightWhite10  db 'knight white > 10',0
00449120 aKnightBlack10  db 'knight black > 10',0
00449134 aPawnWhite8     db 'pawn white > 8',0
00449144 aPawnBlack8     db 'pawn black > 8',0
00449154 aPieceWhite16   db 'piece white > 16',0
00449168 aPieceBlack16   db 'piece black > 16',0
0044917C aPawnRankOneOrE db 'pawn rank one
or eight',0
00449194 aCastlelllegale db 'castle illegale',0
004491A4 aWhiteCaptureRe db 'white capture re',0
004491B8 aBlackCattureRe db 'black catture re',0
```

```
aOneWhiteKingRe db 'ONE white king required',0
aOneBlackKingRe db 'ONE black king required',0
aTooManyWhiteQu db 'Too many white queens',0
aTooManyBlackQu db 'Too many black queens',0
aTooManyWhiteRo db 'Too many white rooks',0
aTooManyBlackRo db 'Too many black rooks',0
aTooManyLightSq db 'Too many light-squared white
bishops',0
aTooManyLight_0 db 'Too many light-squared black
bishops',0
aTooManyDarkSqu db 'Too many dark-squared white
bishops',0
aTooManyDarkS_0 db 'Too many dark-squared black
bishops',0
aTooManyWhiteBi db 'Too many white bishops',0
aTooManyBlackBi db 'Too many black bishops',0
aTooManyWhiteKn db 'Too many white knights',0
aTooManyBlackKn db 'Too many black knights',0
aTooManyWhitePa db 'Too many white pawns',0
aTooManyBlackPa db 'Too many black pawns',0
aTooManyWhitePi db 'Too many white pieces',0
aTooManyBlackPi db 'Too many black pieces',0
aPawnAtRank1Or8 db 'Pawn at rank 1 or 8',0
alllegalCastlin db 'Illegal castling',0
aBlackKingCanBe db 'Black king can be captured',0
aWhiteKingCanBe db 'White king can be captured',0
```

It's this kind of undeniable DNA traces that are left to trace a derivative, even the order of the error messages remained fully intact. This aside the evidences other people reveiled about the origin of Houdini.

The lesson that can be learned from history is that cloners leave (obvious) DNA traces. And such obvious and undeniable DNA traces were not found in Rybka 1.0, see Chapter II.

Chapter V
Additional evidence of Rybka originality

September 2013

When we in September 2012 presented this critical document to various key people of the ICGA it was met by silence until this present day despite various reminders.

Instead, in 2013 the ICGA released another opinion rehashing its (old) arguments from 2011 and virtually ignoring the detailed critique of its evidence.

A quote

We repeat from Section7.2 the summary of this part of the report: ...nearly the entire evaluation function is derived from FRUIT. This includes the formulas for calculating piece- square tables,methods and features of evaluating piece mobility,rookkingfileproximity, rook and queen on the 7thrank,and king safety.

We leave it to readers to study the evidence presented in the investigation report, in Riis's article and in Mark Watkins (2011) rebuttal, and to decide for themselves who they believe.

End quote

Unlike the ICGA Rybka investigators (Zach Wegner & Mark Watkins) our own investigation highlights many important points that Wegner and Watkins missed, such as errors in the reversed engineered material of which some were deliberately withheld from the documents and information by Rajlich that deliberately was withheld by the ICGA to the investigation Panel.

Aside from that, whereas the ICGA Rybka investigators could only find 5-7 differences between the evaluation of Fruit and Rybka, the contra investigation shows a long list.

In short:

Every major evaluation ingredient is coded differently, mobility, king safety, passed pawns, double pawns, backward pawns, Rybka is missing Fruit's late endgame knowledge, Rybka

has a material table different from Fruit. Rybka does not contain Fruit's quad function, Rybka's trapped bishop evaluation is different, rook evaluation is different, bishop pair evaluation different and **most importantly,** Fruit evaluates in stages in a unique way whereas Rybka adds directly to its score (as every other program on the planet does).

Another difference is that Fruit evaluates the white & black pieces in one block of code whereas Rybka evaluates in 2 separate blocks of split code.

Other (minor and key) differences between Fruit and Rybka:

1. Time control is different
2. Fen parsing is different
3. Rybka extracts the mainline from the TT, Fruit via the classic triangle table
4. Rybka 1.0 beta displays a mainline of max 10 plies, Fruit produces much longer variations
5. Rybka uses a bitboard board representation whereas Fruit is mailbox
6. Rybka's pawn value is 3200 (which is unique), Fruit uses the classic value of 100.
7. The order in the evaluations of Fruit and Rybka are not similar
8. Rybka has lazy evaluation, which is absent in Fruit
9. Futility pruning is different between the two programs
10. Rybka uses Late Move Reductions (LMR) whereas Fruit uses history reductions
11. Fruit uses a history table which is absent Rybka.
12. Fruit only has one evaluation table (king safety) while Rybka has many.
13. Rybka does not handle promotions to minor pieces
14. The two programs have a different move format
15. The two programs have different hash table code
16. They have different handling of repetitions and the 50-move rule in search
17. Fruit maintains piece-lists which are not present in Rybka
18. Fruit maintains a pseudo "bitboard" for pawns, Rybka has the real thing.
19. Contrary to Fruit, Rybka needs to update 4 rotations of occupancy bitboard.
20. Contrary to Fruit, Rybka updates a rough estimate of material balance with weights of 1:3:3:5:10 (in the evaluation this value is then corrected by a delta obtained from the material table)

21. Fruit has a 16*16 square mailbox, Rybka has an 8*8
22. The programs have different Zobrist hash keys
23. The programs have different user interface options

We found 36 indisputable differences (some of which are very fundamental), not counting the ones that are debatable such as the PST's and the contents of the TT.

Why didn't Wegner & Watkins come up with this list but only found 5-6-7 differences?

And if Wegner & Watkins had researched the Rybka 2.3.2 version, the one that actually played in an ICGA tournament (and not Rybka 1.0 beta) they would have found that the list of differences was much greater.

Mark Watkins - *I think the case for "copyright infringement" (or plagiarism) of the evaluation function as a whole is quite weighty, particularly when combined with the various other Fruit 2.1 bits that appear here-and-there in Rybka 1.0 Beta.*

So by the standard of Watkins we need to judge the contra evidence **as a whole** (36 points) too and the accusation that Rajlich copied Fruit becomes less and less plausible the longer the list becomes.

Further "**remarkable**" statements of the two ICGA Rybka investigators:

Mark Watkins - *The Rybka 1.0 Beta executable contains no literally copied evaluation code from Fruit 2.1.*

Zach Wegner - *Coming back to Rybka/Fruit, yes, if you look at each example in detail, you can't say there is much hard evidence of direct code copying. As I said to Vas, I'm only completely certain that three characters were copied ("0.0").*

Note - The credit for points 14-21 goes to Richard Vida author of Critter when he was asked to reverse engineer parts of Rybka 1.0 beta that Watkins and Wegner didn't research.

Note - Please find our answer to the ICGA (2013) document here.

Chapter VI
an experiment

Somewhere in 2012 Mark Watkins and I agreed to an experiment. I proposed Mark to take the Fruit source code and change all evaluation values to Rybka values, then measure the percentage difference with similarity tester. Watkins took the challenge and reported a 4% increase.

From Zach's document - Simply put, Rybka's evaluation is virtually identical to Fruit's.

Meaning, if we give Fruit (or Rybka 1.0 beta for that matter) all the same evaluation values then (using Zach's words) Fruit and Rybka are virtually identical. Naturally Watkins liked the idea, enough for him to put considerable time in the idea, for me the outcome was quite an important moment. If suddenly similarity tester would report sky-high numbers then certainly I would have to rethink my position from scratch.

And the Rybka-Fruit similarity went up from 54% to 58%.

Now 58% similarity is a very poor number for 2 **identical** evaluations.

If we take the 58% and compare it with Adam's clone study it doesn't even make it to the top-30 of the suspect engines list.

```
1. Fruit 2.2.1        vs  Loop 2007          84.73% (similarity)
2. Fruit 2.2.1        vs  Loop 10.32f        84.13%
3. Strelka 1.8        vs  Rybka 1.0 Beta     73.11%
4. Fruit 2.1          vs  TogaII 1.0         72.43%
5. Fruit 2.1          vs  Loop 2007          70.81%
6. RobboLito 0.085d1  vs  Houdini 1.00       70.37%
   ................
```

```
29. Glaurung 1.2.1     vs  Alfil 8.1.1        59.47%
30. Rybka 3            vs  Houdini 1.00       59.37%
```

The logical question here is, how can Zach's statement - Simply put, Rybka's evaluation is virtually identical to Fruit's possibly be true?