Combocheck Documentation

About Combocheck

Combocheck is an application for helping find cases of academic misconduct between students in large computer science classes. What it does is pair all the selected submissions with one another, then run a series of algorithms to determine similarity between the submissions in different ways. It does not definitively say whether or not two submissions are guilty of academic misconduct; it merely orders submission pairs by the different similarity metrics, and allows the user to decide for themselves.

After scanning a set of submissions, the ranked submission pairs may be viewed side-by-side in Combocheck, and added to a list of pairs to report for academic misconduct. The pairs can be given optional scores or comments about what in the submissions constitutes academic misconduct. Finally, the pairs can be exported as HTML files that visually demonstrate the differences between the files, for ease in reporting academic misconduct cases to the Office of Student Integrity.

Andrew Wilder, the author of Combocheck, graduated from Georgia Institute of Technology in 2016, specializing in systems, networking and theory. Combocheck started as a project of his in 2015 to see if some algorithms he had learned in classes could be useful in detecting plagiarism between students in the class he was a teaching assistant for. Initially, it only supported one algorithm (edit distance), with very few options for customization at the time. However, its use proved so effective at finding cases of academic misconduct, that he decided to put much more time into Combocheck, and submit it as his master's project. It is currently still being used in the class Andrew taught for.

Questions? Feature suggestions? Bug reports? Please let me know!

andrew.m.wilder@gmail.com

Available Algorithms

Combocheck supports 4 different algorithms:

- Moss
- AST Isomorphism
- Edit Distance
- Token Distance

Each can be individually enabled/disabled for a scan, and some of which have configurable options.

For algorithms that require some form of parsing on a per-language basis, such as Token Distance or AST Isomorphism, generated parse trees are cached for the duration of the scan to minimize computing time. The algorithms themselves are optimized to run in parallel, and do so in several concurrent threads.

The algorithms are all implemented in both Java, and C++. The C++ (native) versions of the algorithms are contained in a dynamic linked library distributed with Combocheck; however, these DLL files must be compiled separately for each architecture and operating system utilizing Combocheck. By default, the native versions are used because they run much faster than the Java versions. However, the Java versions will be used if Combocheck is unable to load the native algorithms DLL.

Moss

This algorithm is based off of Stanford's Measure of Software Similarity, detailed in the paper *Winnowing: Local Algorithms for Document Fingerprinting* by Aiken, Schleimer and Wilkerson.

The way the Moss algorithm works is first some form of normalization is applied to the file, such as renaming all the variables and removing whitespace. Then, all substrings of length K (termed “K-grams” by the paper) in the file are hashed to generate an ordered hash list. Finally, a sliding window of some fixed size is run over all the hashes, taking the new numerical minimum hash value at each step (or none if it is the same) and adding it to a fingerprint. The resulting fingerprints for all the files are compared against one another using the edit distance algorithm.

This algorithm is very fast, but not as accurate as some of the other algorithms. The Moss algorithm in Combocheck has the following configurable options:

- K-gram size
- Winnowing window size
- Normalization type (none, whitespace, variables and whitespace)

Note: If the normalization type “Variables” is selected and a submission file fails to parse due to a syntax error or being written in a language which is not supported by Combocheck, Moss will fall back to the “Whitespace only” normalization type for that file only.

AST Isomorphism

The AST Isomorphism algorithm is based off the Aho-Hopcroft-Ullman algorithm (see *Alogtime Algorithms for Tree Isomorphism, Comparison and Canonization* by Buss) for determining isomorphism between two files' abstract syntax trees.

This algorithm is the fastest in Combocheck, typically completing in less than a second even for hundreds of submissions. AST Isomorphism, unlike the other algorithms in Combocheck, does not produce a metric by which file pairs can be ranked; instead, for each file pair it simply produces either:

- 0 (true) – The files have isomorphic abstract syntax trees.
- 1 (false) – The files do not have isomorphic abstract syntax trees.
- 2 (error) – At least one of the files in this pair failed to produce a parse tree.

This algorithm operates on a per-language basis, and requires that the language used in the submission pairs be supported by Combocheck.

Edit Distance

The Edit Distance algorithm is based off the Wagner-Fischer dynamic programming algorithm for determining the number of insertions, deletions and mutations necessary to transform one array into another. This is the slowest algorithm in Combocheck, but one of the most accurate in determining cases where plagiarism has occurred.

The only option for the Edit Distance algorithm is the file normalization type. If the normalization type “Variables” is selected and a submission file fails to parse due to a syntax error or being written in a language which is not supported by Combocheck, Edit Distance will fall back to the “Whitespace only” normalization type for that file only.

In general, the token distance algorithm performs faster, and is tripped up less by manual obfuscations of the source files due to its removal of comments, and normalization of variable names.

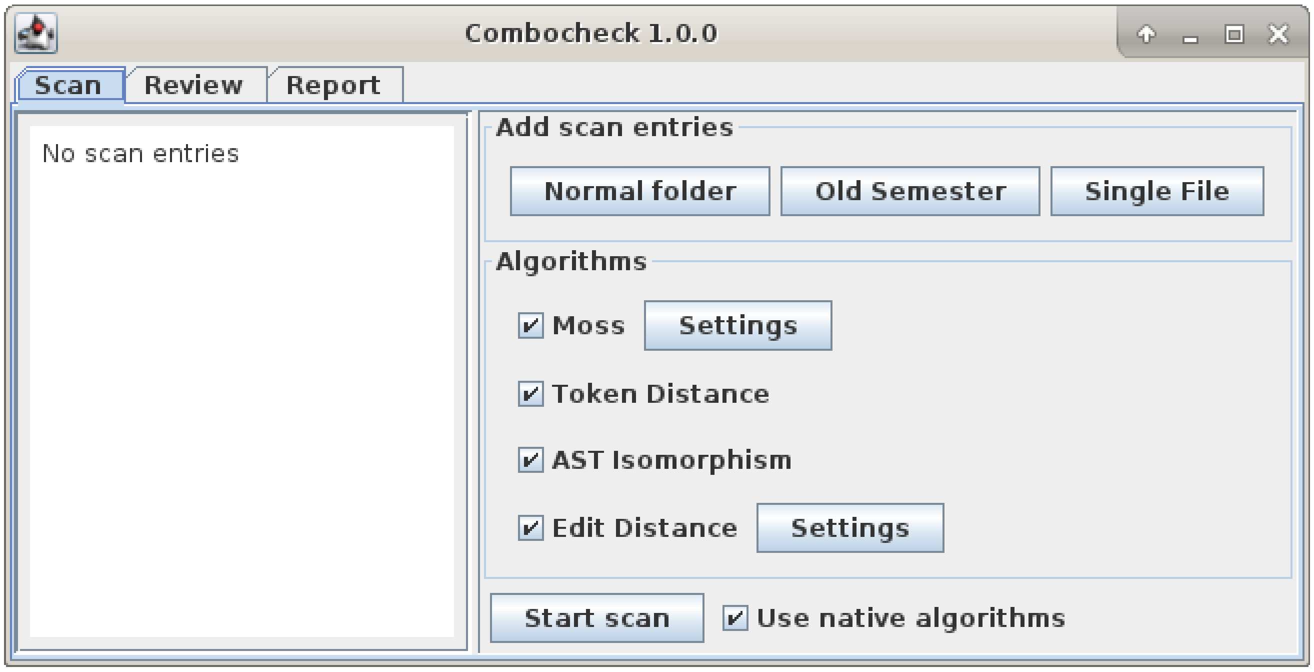
Token Distance

The Token Distance algorithm first parses the file on a per-language basis to generate an array of numerical token IDs corresponding to the different types of tokens encountered while scanning the file, then performs an edit distance over the token ID arrays.

Token distance tends to be the most accurate algorithm in determining cases in which submission pairs are copying off one another, but is significantly slower than Moss, since it is based off the Wagner-Fischer algorithm which is $O(n^2)$. It should still complete in a reasonable amount of time for up to about 300 submissions, but scanning 400+ submissions will take a significant amount of time, and you may want to consider simply using just Moss and AST Isomorphism in that case.

Token distance is a slow algorithm, but it generally performs faster than the regular Edit Distance algorithm, and is tripped up less by manual obfuscations of the source files due to its removal of comments, and normalization of variable names.

Scanning Submissions



No scan entries

☑ Moss

☑ Token Distance

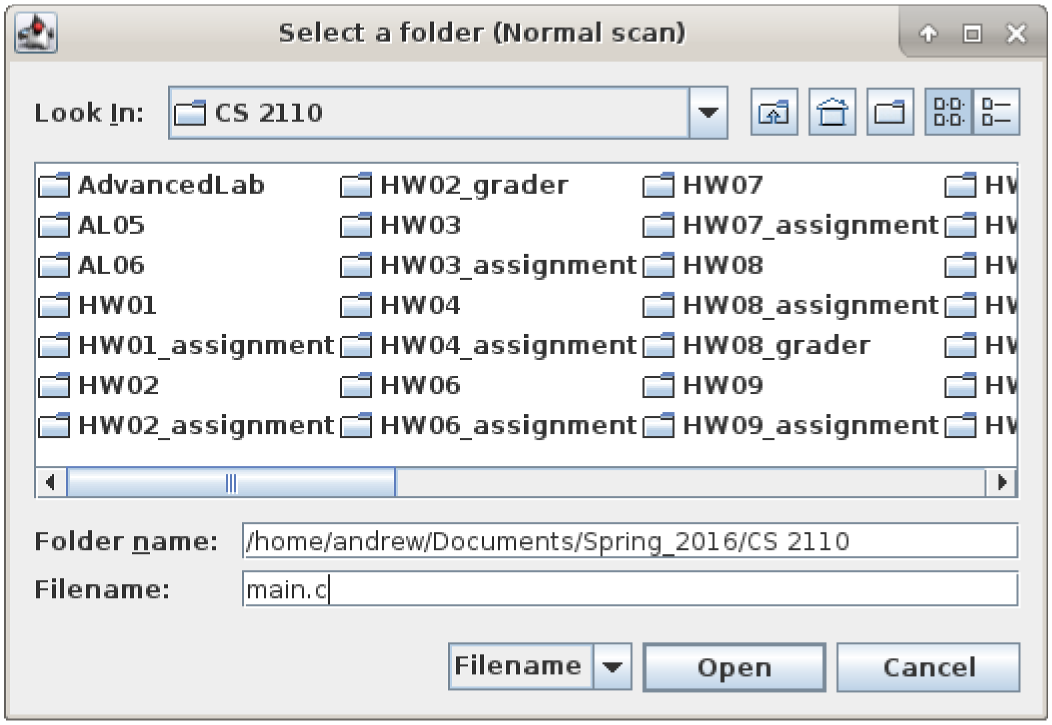
☑ AST Isomorphism

☑ Edit Distance

☑ Use native algorithms

Scanning submissions is done by selecting files or directories to scan under “Add scan entries”, configuring which algorithms are used under “Algorithms”, and then by starting the scan.

Selecting Submission Folders



Folder name: /home/andrew/Documents/Spring_2016/CS 2110

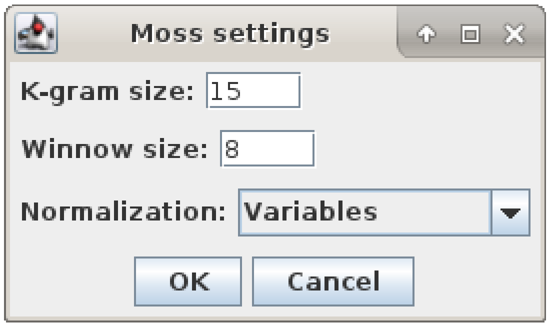
Filename: main.c

Submission folders can be selected using the buttons for “Normal Folder”, “Old Semester”, and “Single File”. The Normal Folder and Single File selections are used to select files which will all be paired up with one another when the scan starts. The Old Semester selection will specify a folder with additional submissions to compare the current semester's files against, but the files in “Old Semester” folders will not be compared against each other.

For “Normal Folder” and “Old Semester”, the files may be located by name, or by regular expression (see the java.util.regex.Pattern documentation for regex syntax)

Tip: In the case that you receive a late submission and want to check it against all the current semester's files, but not have to check those other files against each other again, you can add the new file as a “Single File”, and the current semester as an “Old Semester” to file pairing to only contain the new file.

Configuring Scan Options



K-gram size: 15

Winnow size: 8

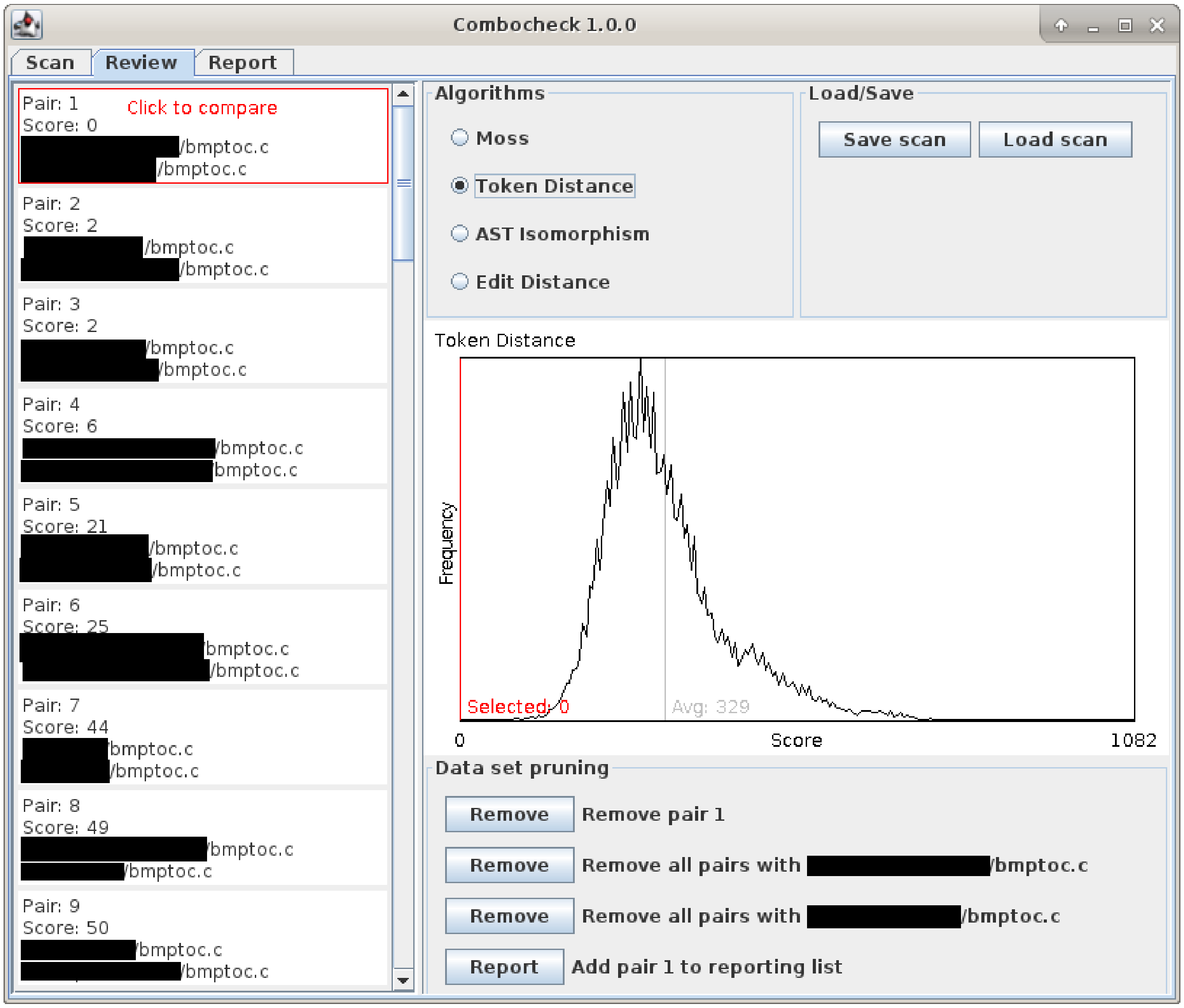
Normalization: Variables

Combocheck's supported algorithms may be individually enabled/disabled, as well as configured. At least one algorithm must be enabled to perform a scan, of course. The Moss algorithm's K-gram and winnowing window sizes can be configured here, and both Moss and Edit Distance have different types of file normalization options available:

- None: The file is not modified at all.
- Whitespace only: All whitespace and tab characters are removed, and all characters are made lowercase.
- Variables: On a per-language basis, all variable names are changed to "VAR", which is useful to catch cases where two submissions have the same code, but different variable names. All whitespace is also removed by this option. If a file fails to parse due to a syntax error or being written in a language not supported by Combocheck, then it will fall back to the "Whitespace only" normalization type.

The checkbox "Use native algorithms" can manually enable or disable use of Combocheck's faster C++ algorithms, if for whatever reason you choose not to use them. If Combocheck was unable to load the native algorithms library on startup, this option will be disabled, and Combocheck will use the Java implementations of the algorithms.

Reviewing Submissions



Pair: 1 Click to compare
Score: 0
[redacted]/bmptoc.c
[redacted]/bmptoc.c

Pair: 2
Score: 2
[redacted]/bmptoc.c
[redacted]/bmptoc.c

Pair: 3
Score: 2
[redacted]/bmptoc.c
[redacted]/bmptoc.c

Pair: 4
Score: 6
[redacted]/bmptoc.c
[redacted]bmptoc.c

Pair: 5
Score: 21
[redacted]/bmptoc.c
[redacted]/bmptoc.c

Pair: 6
Score: 25
[redacted]bmptoc.c
[redacted]/bmptoc.c

Pair: 7
Score: 44
[redacted]bmptoc.c
[redacted]/bmptoc.c

Pair: 8
Score: 49
[redacted]bmptoc.c
[redacted]/bmptoc.c

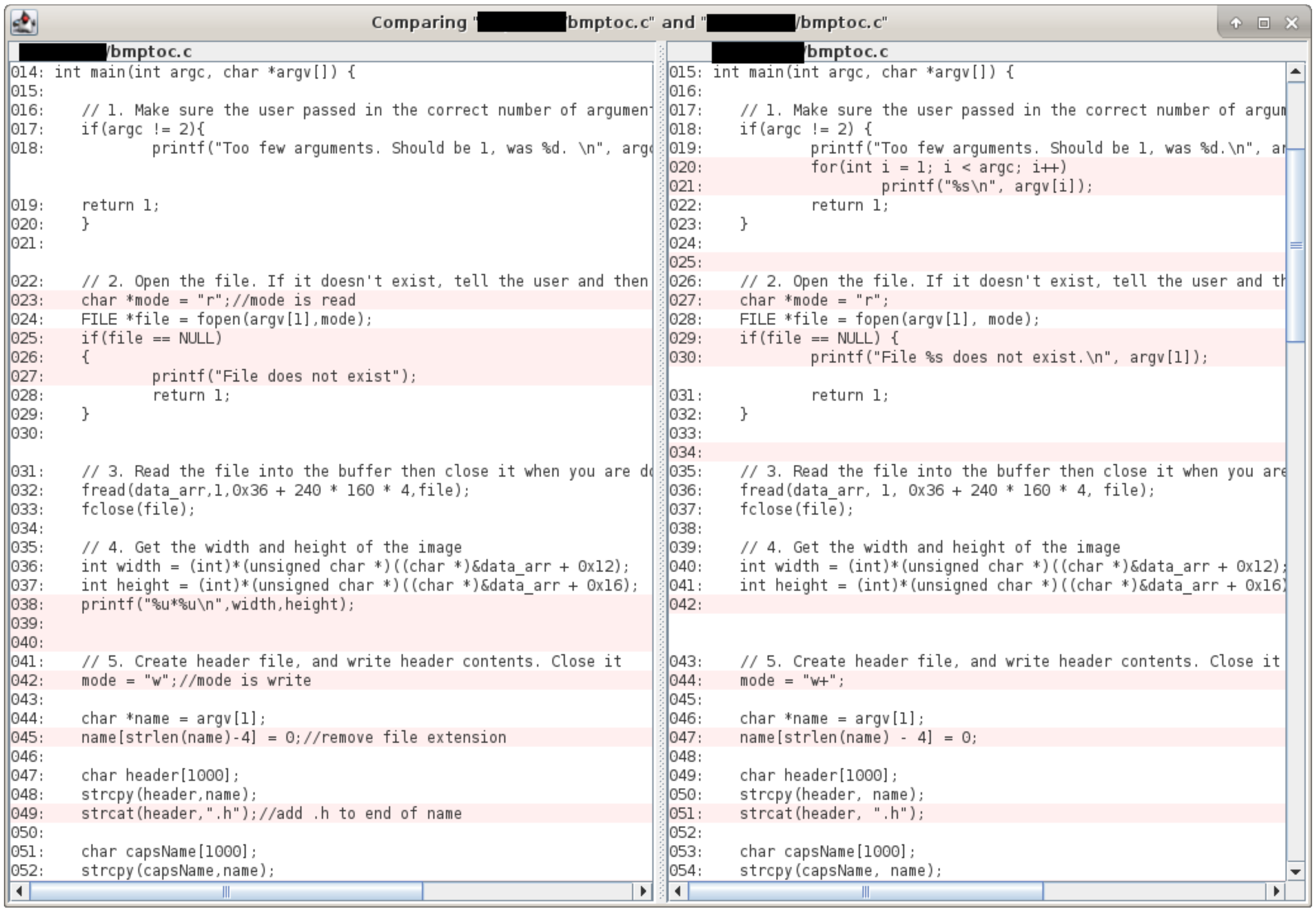
Pair: 9
Score: 50
[redacted]bmptoc.c
[redacted]/bmptoc.c

Algorithms: Moss [✓ Token Distance] AST Isomorphism Edit Distance

Selected: 0
Avg: 329
0 Score 1082

Remove pair 1
Remove all pairs with [redacted]/bmptoc.c
Remove all pairs with [redacted]/bmptoc.c
Add pair 1 to reporting list

Once a scan is completed, you can review the submissions under Combocheck's “Review” tab. Here, you can view the distribution of pair similarities, as well as the ordering of scores by algorithm. Click a pair while it is selected to bring up the comparison dialog and view the files visually. You can also prune the selection by removing individual pairs, or all pairs containing a certain file (for instance, in the case someone submitted a blank template file). Click the report button to add the selected pair to the list of pairs for reporting. You can also save or load scans, which will include pairs selected for reporting, and their annotations.



Comparing "[redacted]bmptoc.c" and "[redacted]/bmptoc.c"

[redacted]/bmptoc.c

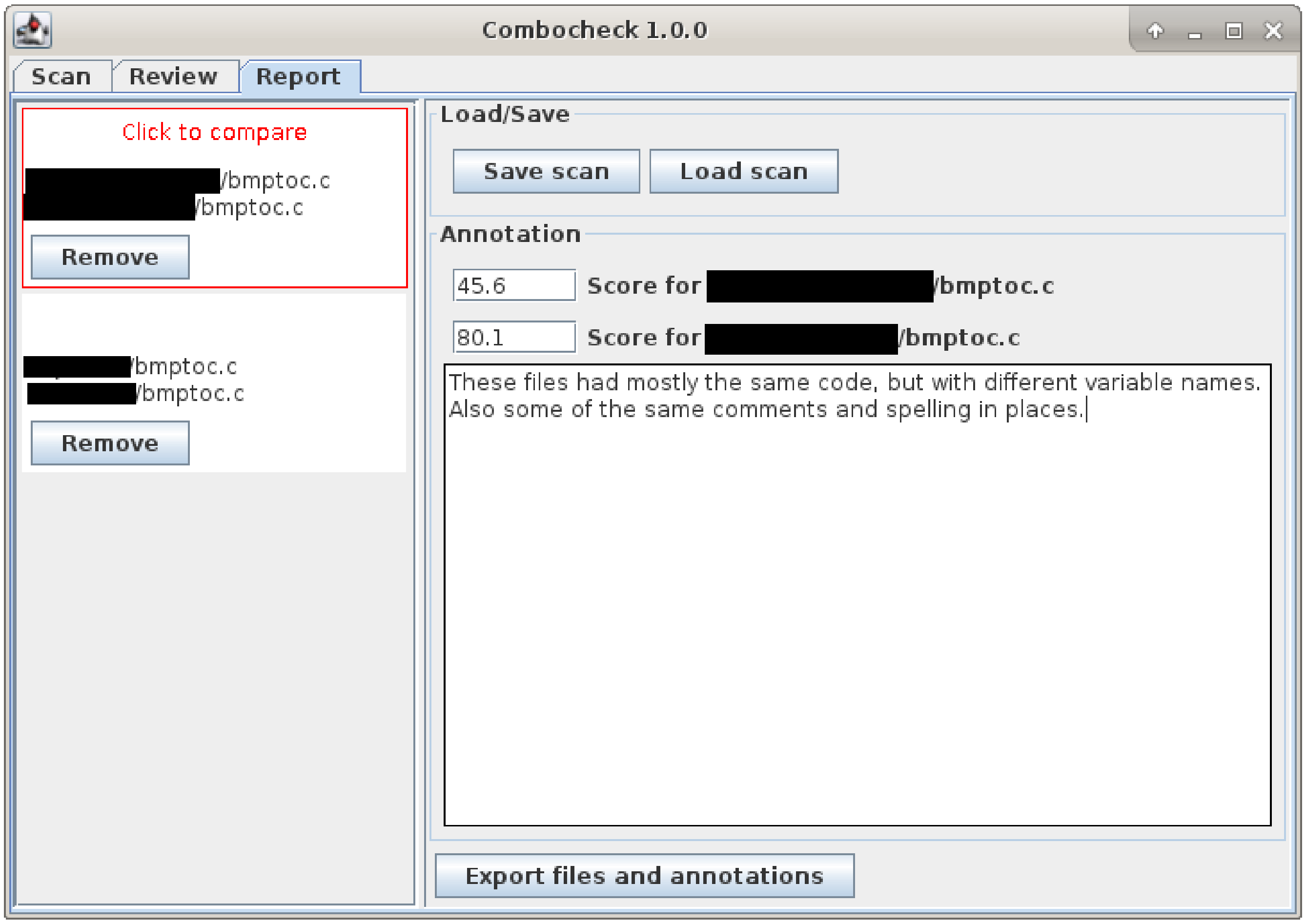
```
int main(int argc, char *argv[]) {

    // 1. Make sure the user passed in the correct number of argumen
    if(argc != 2){
            printf("Too few arguments. Should be 1, was %d. \n", arg
    return 1;
    }

    // 2. Open the file. If it doesn't exist, tell the user and then
    char *mode = "r";//mode is read
    FILE *file = fopen(argv[1],mode);
    if(file == NULL)
    {
            printf("File does not exist");
            return 1;
    }

    // 3. Read the file into the buffer then close it when you are d
    fread(data_arr,1,0x36 + 240 * 160 * 4,file);
    fclose(file);

    // 4. Get the width and height of the image
    int width = (int)*(unsigned char *)((char *)&data_arr + 0x12);
    int height = (int)*(unsigned char *)((char *)&data_arr + 0x16);
    printf("%u*%u\n",width,height);


    // 5. Create header file, and write header contents. Close it
    mode = "w";//mode is write

    char *name = argv[1];
    name[strlen(name)-4] = 0;//remove file extension

    char header[1000];
    strcpy(header,name);
    strcat(header,".h");//add .h to end of name

    char capsName[1000];
    strcpy(capsName,name);
```

[redacted]/bmptoc.c

```
int main(int argc, char *argv[]) {

    // 1. Make sure the user passed in the correct number of argum
    if(argc != 2) {
            printf("Too few arguments. Should be 1, was %d.\n", ar
            for(int i = 1; i < argc; i++)
                    printf("%s\n", argv[i]);
            return 1;
    }


    // 2. Open the file. If it doesn't exist, tell the user and th
    char *mode = "r";
    FILE *file = fopen(argv[1], mode);
    if(file == NULL) {
            printf("File %s does not exist.\n", argv[1]);

            return 1;
    }


    // 3. Read the file into the buffer then close it when you are
    fread(data_arr, 1, 0x36 + 240 * 160 * 4, file);
    fclose(file);

    // 4. Get the width and height of the image
    int width = (int)*(unsigned char *)((char *)&data_arr + 0x12);
    int height = (int)*(unsigned char *)((char *)&data_arr + 0x16)

    // 5. Create header file, and write header contents. Close it
    mode = "w+";

    char *name = argv[1];
    name[strlen(name) - 4] = 0;

    char header[1000];
    strcpy(header, name);
    strcat(header, ".h");

    char capsName[1000];
    strcpy(capsName, name);
```

The comparison dialog will automatically line up code so that differences are easy to spot, and highlight lines unique to each file in red. The comparison dialog is accessible from both Combocheck's “Review” tab (by clicking an already-selected file pair), and from the “Report” tab (by click an already-selected reporting pair).

Reporting Submissions



[redacted]/bmptoc.c
[redacted]/bmptoc.c

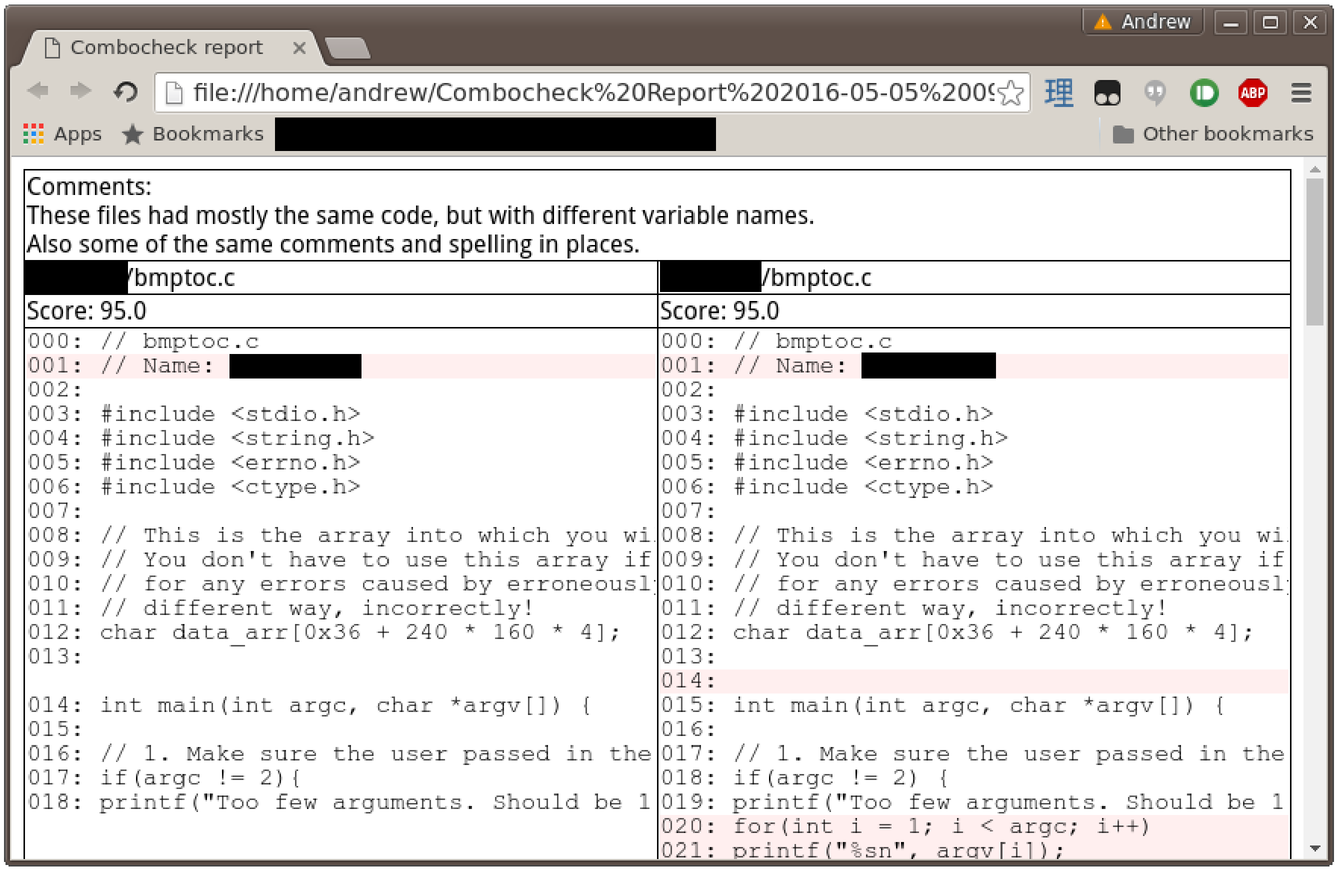
[redacted]/bmptoc.c
[redacted]/bmptoc.c

45.6 Score for [redacted]/bmptoc.c

80.1 Score for [redacted]/bmptoc.c

These files had mostly the same code, but with different variable names. Also some of the same comments and spelling in places.

Submission pairs added to the reporting tab can be annotated with optional scores they would have received, and a message detailing what you believe constitutes academic misconduct between the files. After annotating files, you can save the scan from this screen as well (saving from either this screen or the "Review" screen does not matter, both will save the annotations). Once you are satisfied with the selected file pairs and annotations, the pairs can then be exported as a report.



Combocheck report

file:///home/andrew/Combocheck%20Report%202016-05-05%200(

Apps Bookmarks [redacted] Other bookmarks

Comments:
These files had mostly the same code, but with different variable names.
Also some of the same comments and spelling in places.

[redacted]/bmptoc.c

Score: 95.0

```
000: // bmptoc.c
001: // Name: [redacted]
002:
003: #include <stdio.h>
004: #include <string.h>
005: #include <errno.h>
006: #include <ctype.h>
007:
008: // This is the array into which you wi
009: // You don't have to use this array if
010: // for any errors caused by erroneousl
011: // different way, incorrectly!
012: char data_arr[0x36 + 240 * 160 * 4];
013:

014: int main(int argc, char *argv[]) {
015:
016: // 1. Make sure the user passed in the
017: if(argc != 2){
018: printf("Too few arguments. Should be 1
```

[redacted]/bmptoc.c

Score: 95.0

```
000: // bmptoc.c
001: // Name: [redacted]
002:
003: #include <stdio.h>
004: #include <string.h>
005: #include <errno.h>
006: #include <ctype.h>
007:
008: // This is the array into which you wi
009: // You don't have to use this array if
010: // for any errors caused by erroneousl
011: // different way, incorrectly!
012: char data_arr[0x36 + 240 * 160 * 4];
013:
014:
015: int main(int argc, char *argv[]) {
016:
017: // 1. Make sure the user passed in the
018: if(argc != 2) {
019: printf("Too few arguments. Should be 1
020: for(int i = 1; i < argc; i++)
021: printf("%sn", argv[i]);
```

The exported HTML per submission pair is of a similar format to the comparison dialog, though this also contains the annotation for the pair as well. The exported folder also includes the files themselves, so other tools such as ExamDiff Pro can be used to compare the submissions as well if you prefer.

Credits

Combocheck uses the following 3rd-party libraries:

- ANTLR 4.5.2
- Apache Commons Lang 3.4