Anti-plagiarism Project

Eric Scott Freeman

Abstract—This project integrates several anti-plagiarism tools into Autograder, software used by the University of Stavanger to automatically grade students' programming assignments. The three tools used were good at finding duplicate pieces of code, but it is still up to the instructor to determine whether or not it could be considered plagiarism.

Index Terms—anti-plagiarism, plagiarism, fingerprinting, stylometry, Moss, JPlag, dupl, Autograder.

1 Introduction

The University of Stavanger uses an application called Autograder, written by Heine Furubotten, to automatically grade students' programming assignments. Autograder works with GitHub. Whenever a student pushes their code to GitHub, Autograder will pull a copy of the code and run a series of tests on it. Autograder does not currently have support for plagiarism detection. The goal of this project is to integrate a few anti-plagiarism tools into Autograder, thereby helping professors save time.

2 Related Work

Unfortunately software plagiarism is a problem both in the classroom and in the workplace. A number of applications have been created to help detect this problem. While these tools can detect similarities in programs, the flagged files must still be manually examined to determine whether or not code was plagiarized.

There are several different general techniques that are used to look for plagiarism. The tools analyzed in this project used either fingerprinting or stylometry.

2.1 Fingerprinting

Several tools use a technique called fingerprinting to detect plagiarism. In fingerprinting algorithms, hashes of n-grams, substrings that are n characters, are saved and compared to help find plagiarism. Not all hashes are stored due to the large number that would be produced.

2.1.1 Winnowing

Moss uses a technique called winnowing to select which hashes to save [1]. In the winnowing algorithm, a window, selection of contiguous hashes, is used to help select which hashes to save. The smallest hash from a window is saved, and then the window moves one hash over. The smallest hash from the next window is often the smallest hash from the previous window. If so it is not saved again. Figure 1 shows an example of how winnowing works. The orange box represents the shifting window. The green box shows whenever a new hash is saved.

2.1.2 Running-Karp-Rabin Greedy-String-Tiling

JPlag uses Running-Karp-Rabin Greedy-String-Tiling (RKS-GST) to compare hashes of code in plagiarism detection [2].

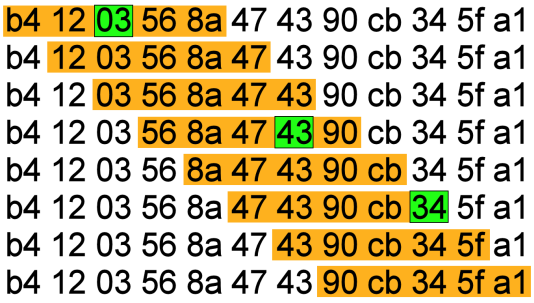


Fig. 1. Winnowing.

RKS-GST was originally used in YAP3, another plagiarism detection tool. In RKS-GST, the Greedy String half of the algorithm forms pairs of substrings, each from a different string. Then the Karp-Rabin half of the algorithm hashes each substring in the pair [3]. This is done to help detect code reordering.

2.2 Stylometry

Another approach is to use code stylometry, which analyzes the style of writing or coding.

2.2.1 Abstract Syntax Trees

Caliskan-Islam, et al. use abstract syntax trees (ASTs) to compare the styles of authors [4]. Things that are easily changed in code, such as variable names, become leaves in the AST, while the structure of the tree is harder to change [4]. Figure 2 shows an abstract syntax tree of the code in Figure 3. Note how the leaves, or circular nodes, in Figure 2 are variable names, constants, and a function name.

Michal Bohuslávek's dupl application uses ASTs to find similarities in code [5]. It looks for any copies of code, not just plagiarism. So if a piece of code is duplicated even in the same file, it will test positive.

2.3 Supported Languages

Fingerprinting and string comparison techniques can be used to analyze source code written in languages other than their officially supported languages. Moss can analyze Go code, even though it is not technically supported. Since ASTs need to parse the code, applications which use them are stricter on which languages they can analyze. For example

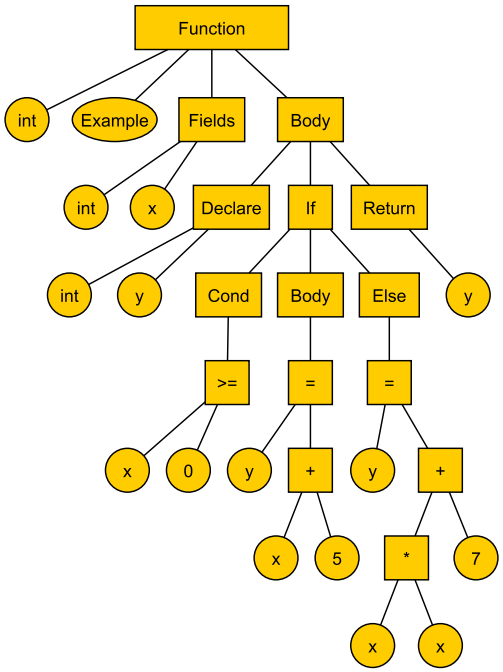


Fig. 2. Abstract syntax tree.

```
int Example(int x) {
  int y;

  if (x >= 0) {
    y = x + 5;
  }
  else {
    y = x * x + 7;
  }

  return y;
}
```

Fig. 3. Example code.

dupl only supports code written in Go. Table 1 shows the languages officially supported by several anti-plagiarism tools.

Tool	Java	Go	C	C++	C#	Python	others
Moss	✓		✓	✓	✓	✓	✓
JPlag	✓		✓	✓	✓		✓
Plaggie	✓						
SIM	✓		✓				✓
dupl		✓					

TABLE 1
Officially supported languages by various tools

3 Design

3.1 Selection of tools

Moss was the first choice in anti-plagiarism tools, because it supports a large number of languages and is a mature piece of software. Two other tools were added later: JPlag and dupl. Like Moss, JPlag supports many languages. While dupl is less than a year old, it uses ASTs. These three tools allow the project to use three different techniques for finding plagiarism: winnowing, RKS-GST, and ASTs.

3.2 Process

Since Autograder was written in Go, this project was also written in Go. Later it was decided to make this project a standalone application that Autograder will call, since another university has expressed interest in using it. The program calls each of the anti-plagiarism tools and stores the results. A separate Go package was written for each anti-plagiarism tool. The packages implement a common interface which creates the commands to send to the tools and formats the results. The interface will allow other anti-plagiarism tools to be easily added later.

The anti-plagiarism application typically runs as a service and accepts gRPC, a remote procedure call framework, requests. A request consists of the GitHub organization, the GitHub authorization token, an array of student repository names, and an array of lab assignment names and programming languages. Requests can also be sent from the command line.

The `golint` application was run against the code for suggestions on making the code follow Go coding conventions. The `go fmt` command was run to clean up the whitespace in the code.

3.2.1 *Calling the anti-plagiarism tools*

Before calling the anti-plagiarism tools, first the students' code is pulled from GitHub to the Autograder server. The directory structure consists of a base directory containing subdirectories for each class. Each class is a GitHub organization. Inside each class, there are directories for each student, which have directories for each assignment. See figure 4. This is similar to how Autograder stores the files in GitHub, so pulling the code for the anti-plagiarism project is simplified.

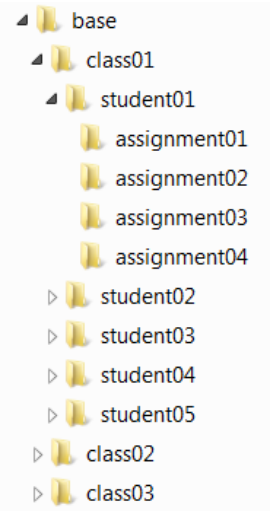


Fig. 4. Directory structure.

Next the commands are created for each anti-plagiarism tools based on the location of the code, the language in

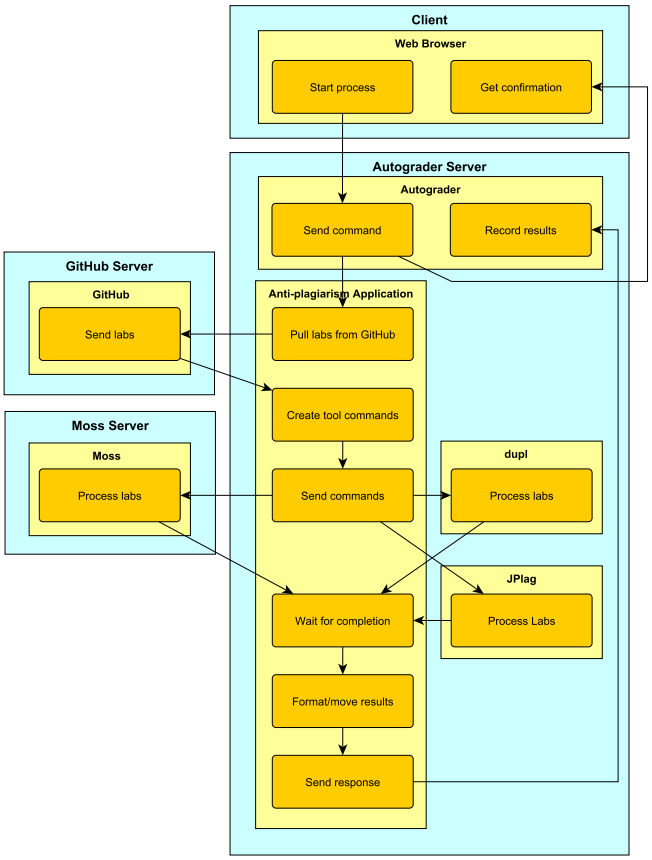


Fig. 5. Process

which the code was written, the threshold of the tool, and various other parameters. The commands are then sent to their respective tools. Each student's work for an assignment is compared against the other students' work for that particular assignment in that class instead of all the assignments for that class or all the assignments for all classes. This keeps the number of files being compared from growing too large.

The anti-plagiarism application waits for the results to come back from each of the tools. When they have all finished, the program will collect and store the resulting HTML files in the location specified. It also examines the HTML files for the highest incidents of plagiarism for each student. The highest values are written to a JSON file. Finally the program sends a gRPC response back to the calling application indicating if it was successful or not.

3.3 Integrating with Autograder

A few new fields needed to be added to the Autograder's Bolt database. Bolt stores entries as a key/value pair. Each lab assignment entry in each class needed to store the language the assignment is written in. Also each lab assignment entry for each student/group needed to store the results from each of the anti-plagiarism tools.

The user interface was also updated to allow teachers to assign a programming language to each lab (figure 6), to allow teachers to initiate the anti-plagiarism process (figure 7), and to show the results of the process. When the Autograder web service receives a request to start the anti-plagiarism process, it creates the gRPC command. If it was successful it sends the request and informs the user that the request was sent. Since it can take at least several minutes to process the data, it is best to inform the user immediately that things might take a while. Otherwise they could think that it is not working. Then Autograder starts a Go routine (separate thread) to call the anti-plagiarism software and process the results.

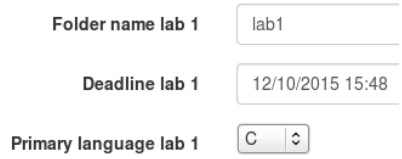


Fig. 6. Selecting language.

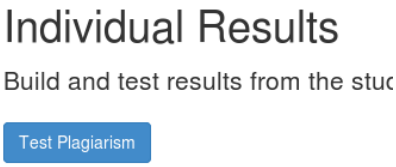


Fig. 7. Start anti-plagiarism test.

The results are stored in the database and also displayed to the teachers through the teacher panel in Autograder. The teacher panel shows a table of all the students in the class and their assignments. If any of the anti-plagiarism tools find evidence of plagiarism, the corresponding cell in the table will be colored a shade of red. The more tools finding plagiarism, the deeper the color. From the teacher panel, teachers go to a results page to look at more details about each lab. This page shows the results of each anti-plagiarism tool and buttons to show the plagiarized code. See figure 8.

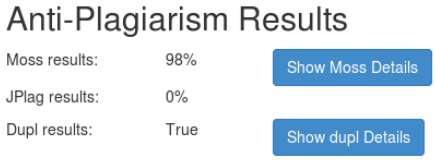


Fig. 8. Results for one lab.

While adding the new functionality to Autograder, care was taken not to break any existing functionality. Unit tests were rerun to check that nothing was broken.

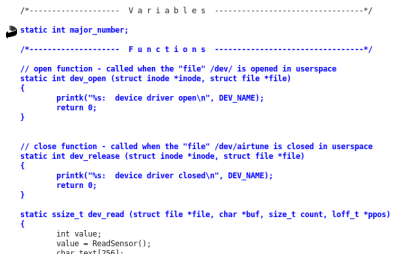
4 Results

4.1 Choosing threshold values

Choosing threshold values took some consideration plus trial and error. Since the threshold value for Moss is the number of files duplicate code can appear in and still be considered plagiarism, it was straightforward to find a value for Moss. The test class consisted of seventeen students and six groups. Therefore the Moss threshold would need to be less than six, because all of the groups would have code provided by the instructor. So the value chosen was five.

For JPlag and dupl, the threshold represents the number of tokens to compare. It was not known how much data a

token represents in JPlag, but luckily the output included the number of tokens in each match. In figure 9 it is shown that 20 tokens can match short functions that only print a value. In figure 10 it is shown that 30 tokens can match a long list of variable assignments. Figure 11 shows that 40 tokens will match pieces of code that are more substantial. Therefore a JPlag threshold of 35 was chosen to get better matches.



```
/*------------------- V a r i a b l e s --------------------------------*/
static int major_number;
/*------------------- F u n c t i o n s --------------------------------*/

// open function - called when the "file" /dev/ is opened in userspace
static int dev_open (struct inode *inode, struct file *file)
{
        printk("%s:  device driver open\n", DEV_NAME);
        return 0;
}


// close function - called when the "file" /dev/airtune is closed in userspace
static int dev_release (struct inode *inode, struct file *file)
{
        printk("%s:  device driver closed\n", DEV_NAME);
        return 0;
}

static ssize_t dev_read (struct file *file, char *buf, size_t count, loff_t *ppos)
{
        int value;
        value = ReadSensor();
```

Fig. 9. JPlag match with 20 tokens.

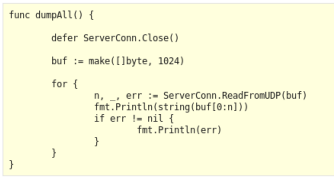
```
        else
                return bufLen;
}

// define which file operations are supported
struct file_operations dev_fops =
{
        .owner  =       THIS_MODULE,
        .llseek =       NULL,
        .read           =       dev_read,
        .write  =       NULL,
        //.readdir      =       NULL,
        .poll           =       NULL,
        //.ioctl        =       NULL,
        .mmap           =       NULL,
        .open           =       dev_open,
        .flush  =       NULL,
        .release        =       dev_release,
        .fsync  =       NULL,
        .fasync =       NULL,
        .lock           =       NULL,
};


// initialize module
static int __init dev_init_module (void)
{
        ccount = 0;
        major_number = register_chrdev( 0, DEV_NAME,
```

Fig. 10. JPlag match with 30 tokens.

```
                bufLen = write_size;
                return write_size;
        }
}

// define which file operations are supported
struct file_operations dev_fops =
{
        .owner          =       THIS_MODULE,
        .llseek         =       NULL,
        .read           =       dev_read,
        .write          =       dev_write,
        //.readdir      =       NULL,
        .poll           =       NULL,
        //.ioctl        =       NULL,
        .mmap           =       NULL,
        .open           =       dev_open,
        .flush          =       NULL,
        .release        =       dev_release,
        .fsync          =       NULL,
        .fasync         =       NULL,
        .lock           =       NULL,
};


// initialize todule
static int __init dev_init_module (void)
{

        major_number = register_chrdev( 0, DEV_NAME, &dev_fops);

        if (major_number < 0)
        {
                printk(KERN_ALERT "Registering char device failed with %d\n", major_number);
                return major_number;
        }

        printk(KERN_INFO "I was assigned major number %d. To talk to\n", major_number);
        printk(KERN_INFO "the driver, create a dev file with\n");
        printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n", DEV_NAME, major_number);
        printk(KERN_INFO "Try various minor numbers. Try to cat and echo to\n");
        printk(KERN_INFO "the device file.\n");
        printk(KERN_INFO "Remove the device file and module when done.\n");
```

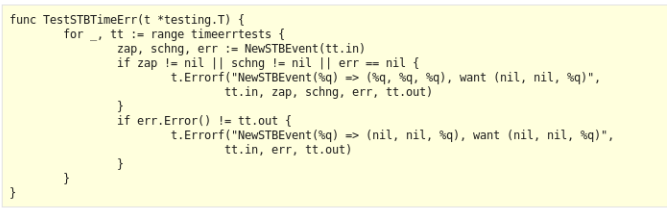
Fig. 11. JPlag match with 40 tokens.

Unfortunately dupl does not provide the number of tokens in a match. So the threshold value needed to be adjusted several times before some of the less detailed results were removed. Figure 12 and figure 13 show examples of matches when the threshold was set to 50 and 75 (up from the default value of 15). 75 was chosen for the final dupl threshold.



```
func dumpAll() {

        defer ServerConn.Close()

        buf := make([]byte, 1024)

        for {
                n, _, err := ServerConn.ReadFromUDP(buf)
                fmt.Println(string(buf[0:n]))
                if err != nil {
                        fmt.Println(err)
                }
        }
}
```

Fig. 12. dupl match when threshold was 50 tokens.

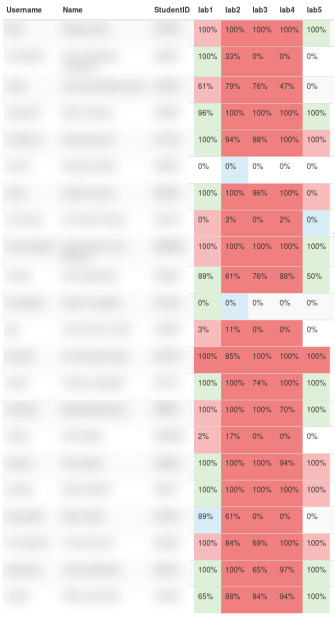


```
func TestSTBTimeErr(t *testing.T) {
        for _, tt := range timeerrtests {
                zap, schng, err := NewSTBEvent(tt.in)
                if zap != nil || schng != nil || err == nil {
                        t.Errorf("NewSTBEvent(%q) => (%q, %q, %q), want (nil, nil, %q)",
                                tt.in, zap, schng, err, tt.out)
                }
                if err.Error() != tt.out {
                        t.Errorf("NewSTBEvent(%q) => (nil, nil, %q), want (nil, nil, %q)",
                                tt.in, err, tt.out)
                }
        }
}
```

Fig. 13. dupl match when threshold was 75 tokens.

4.2 Results in Autograder

Figures 14 and 15 show the individual lab results and group lab results respectively for the test class. In labs one, five, and six, the students programmed in C. They programmed in Go for the remaining labs. All of the test code for the labs was written in Go. Light pink indicates that one tool found duplicate pieces of code, and darker pink indicates two tools found duplicates. Three tools finding results will not occur because there is no overlap in the languages that JPlag and dupl can test.



Username	Name	StudentID	lab1	lab2	lab3	lab4	lab5
			100%	100%	100%	100%	100%
			100%	33%	0%	0%	0%
			61%	79%	76%	47%	0%
			96%	100%	100%	100%	100%
			100%	94%	98%	100%	100%
			0%	0%	0%	0%	0%
			100%	100%	96%	100%	0%
			0%	3%	0%	2%	0%
			100%	100%	100%	100%	100%
			89%	61%	76%	88%	50%
			0%	0%	0%	0%	0%
			3%	11%	0%	0%	0%
			100%	85%	100%	100%	100%
			100%	100%	74%	100%	100%
			100%	100%	100%	70%	100%
			2%	17%	0%	0%	0%
			100%	100%	100%	94%	100%
			100%	100%	100%	100%	100%
			89%	61%	0%	0%	0%
			100%	84%	69%	100%	100%
			100%	100%	65%	97%	100%
			65%	88%	94%	94%	100%

Fig. 14. Individual lab results. Instructors and TAs have accounts too.

Group	Members	lab6	lab7
		100%	100%
		100%	100%
		100%	100%
		100%	100%
		100%	100%
		100%	100%

Fig. 15. Group lab results.

5 Analysis

At first glance, it appeared as if all the students that submitted work plagiarized code, but that would be a very unlikely scenario. Most of the duplicates appear to come from code provided to the students. In labs one and five, little or no code was provided to the students. Therefore there were much fewer detections in these two assignments.

Getting three separate tools to behave similarly using a common interface was a bit difficult. Moss was the original anti-plagiarism tool included in the project before it was decided to add other tools. Moss's threshold says to ignore code that is similar in a certain number of files. So if the threshold is ten and a piece of code appears at least ten times, it is not considered plagiarism. This is very helpful when an instructor gives students a partially completed file or a framework to follow. Since the threshold does not specify the length of code to consider for plagiarism, sometimes relatively short sections of code show up in the results. Unfortunately if a student's lab directory has subdirectories, Moss will say that similar code in different subdirectories is plagiarism.

JPlag can produce false positives in two ways in this project. First, JPlag remembers the results from previous executions. So if a student uses a piece of code in one lab and then reuses the same piece of code in another lab, JPlag can detect this as plagiarism. The second way is JPlag is not ignoring code provided by the instructor. While it is possible to tell JPlag to ignore certain files, this would break the common interface shared among the tools. Also it would require the instructors to format the assignments in such a ways that all of the instructor provided code is separate from all of the student code.

Dupl was not designed to detect plagiarism, only duplicate pieces of code. So it will detect all the instructor provided code as plagiarism. It can even detect two pieces of code in the same file. For example the two functions in figure 16 are from the same file and are shown to be duplicates.

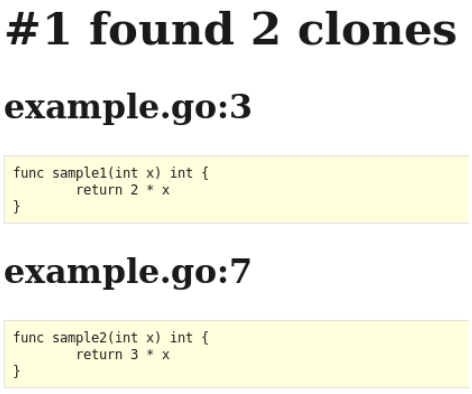


Fig. 16. Sample dupl results from same file.

Since dupl is only looking for duplicate code and not plagiarism, it does not provide a number indicating the percentage of code it thinks is plagiarized. It also places all the results in a single HTML file rather than splitting them up into separate files.

Adjustments were made to the thresholds in attempt to clean up the results shown in figures 14 and 15. Labs one and five improved some, but there was too much provided code in the other labs for cleaner results.

6 Future work

Currently the anti-plagiarism application is only intended to be called locally, so there is no encryption enabled in the gRPC calls. If it is to be called remotely, then transport security needs to be added in the anti-plagiarism application and in Autograder. If the application was called remotely, the result files would need to be sent to the other machine. There are also some hard-coded values in Autograder that should be environment variables or in a configuration file, such as the address and port of the anti-plagiarism application.

Additional anti-plagiarism tools could be added later, but focusing on the use of one tool could be a better option. When using a common interface to access the different tools, some of the fine tuning is lost.

Autograder does not do anything with the percentages other than display them. It could have its own threshold and only display results above a certain percentage.

7 Conclusion

Due to the high number of false positives found during testing, it is probably better to consider the results from this application as supplemental to any suspicions an instructor may have already. It would be tedious to examine all of the results. The tools used are very good at finding similar pieces of code, but there are legitimate reasons for having duplicate code, such as code distributed by the instructor or students reusing their own code. Use of this application is useful if an instructor suspects plagiarism and uses this to verify their suspicions.

Results could be improved in future classes if all instructor provided code resided in separate repositories. This would greatly reduce the number of false positives. Then the work in the students' repositories would hopefully be their own work and not code copied from someone else.

Acknowledgments

The author would like to thank Hein Meling for his advice and guidance during this project.

References

[1] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *In Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pp. 76 – 85, ACM, 2003.

[2] L. Prechelt, G. Malpohl, and M. Philippsen, "Jplag: Finding plagiarisms among a set of programs," *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 1016 – 1038, 2002.

[3] M. Wise, "Yap3: Improved detection of similarities in computer program and other texts," in *SIGCSE '96 Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, pp. 130–134, 1996.

[4] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 255–270, USENIX Association, 2015.

[5] M. Bohuslávek, "dupl." https://github.com/mibk/dupl. Accessed: 2015-09-15.

Eric Scott Freeman is currently a master's student at the University of Stavanger. He received his Bachelor of Science degree in Computer Science from Midwestern State University in 2004. He has also worked as a software developer for several companies including RadioShack, Cisco Systems, and TelStrat.