Abstract—One aspect of software archaeology is retracing (part of) the build environment that was used to compile the binary. The problem is that much of the information about the build-environment gets lost after compilation or due to stripping. The approach taken in this paper is to statistically analyse the distribution of the opcodes in a binary. Opcode statistics are already proven to be effective at detecting metamorphic malware. Work has been done to answer the research question: "Can opcode frequencies be useful for determining the build environment of a binary?".

A collection of binaries were compiled with 6 different optimisation flags and 8 different GCC versions. Single opcodes and opcode combinations (2-grams) were analysed. Statistical differences in opcode frequencies were then measured. The opcode combinations show a slightly stronger relationship as opposed to single opcodes. Statistically, the relationships are weak for the different versions but moderate for the optimisation flags. But however weak, patterns are visible and detectable differences do occur. Looking at the success of detecting metamorphic software using opcode frequencies, there is at least ground for further research. By seeing if a machine learning can be applied to detect compiler versions and/or compiler flags.

I. INTRODUCTION: MOTIVATION

With legacy software there are cases that source code or documentation of the software get lost. All that is left is a binary for which it is unclear on exactly what it does, how it works or how it was designed. Recovering design information and functionality from legacy software can be called software archaeology [1].

One aspect of software archaeology is retracing (part of) the build environment that was used to compile the binary. Being able to retrace the build environment may also have its uses in similar fields such as forensics, reverse engineering and compliance engineering. The problem is that much of the information about the build-environment gets lost after compilation or due to stripping. Used toolchains, version of the compiler and compiler flags are essential parts of the build environment that get lost after compilation. However, different versions of compilers and different compiler optimization flags should generate (slightly) different binary files.

The approach taken in this paper is to statistically analyse the distribution of the opcodes in a binary. An opcode (short for operation code) is used to specify in a machine instruction what operation needs to be performed by the CPU. Opcode instructions can differ depending on the instruction set architecture of the CPU. Most executables are first written in a higher level programming language such as Python or C before they are translated to machine language (aimed for the target computer architecture). This process of translating from higher level code to machine code is called compiling. A binary executable is, in essence, a collection or container of these opcodes (along with strings constants, variable declarations and others).

Opcode statistics are already proven to be effective at detecting metamorphic malware. This is done by training machine learning classifiers to distinguish between goodware and malware. The goal of this paper is to determine if there is potential to apply the same techniques to extract information about the build-environment of the binary. The scope will be the on different versions of GCC (GNU Compiler Collection) and different optimisation flags.

Section 1A will describe the research questions and section 1B the related work regarding the subject. Section 2 covers the methods used to conduct the experiments and the results of these experiments are covered in section 3. A discussion of the results is given in section 4 and the conclusion in section 5, along with some ideas for future work. The appendixes can be found in section 6.

A. Research questions

The main research question is as follows:

"Can opcode frequencies be useful for determining the build environment of a binary?"

This research question is divided into four sub-questions:

1) How significant are the differences in the opcode frequencies when using different compiler versions?
2) How significant are the differences in the opcode frequencies when using different compiler flags?
3) What opcodes are responsible for the differences in the opcode frequencies?
4) Are differences significant enough to detect what compiler flag or version is used for a binary?
B. Related work

Much research has been done on using opcode statistics on malware.

Bilar [2][3] measured the distribution of opcodes on a collection of goodware and a collection of malware. This malware collection consisted out of 7 different malware classes (viruses, bots, rootkits, etc). The goal was to find out whether there is a statistically significant difference in the opcode frequency between the goodware and the seven malware classes. The mov and push codes were the most common opcodes in all cases, but variations in the frequencies of appearance could be seen. The final conclusion was that the less common opcodes (such as int, imul, bt, etc) show the largest variation in frequency and gave the strongest predictions, which could explain 12-63% of the variation. This affirmed that opcodes statistics can be useful in representing binaries.

Santos et al [4] used opcode frequencies to detect malware variants. The best results were measured when using opcode sequences of 2 (instead of using a single opcode). The Mutual Information (MI) equation was used to measure the statistical dependence between an opcode and malware. Weights were then applied to these opcodes so a feature vector could be build from the executables. It was able to identify and distinguish malware variants from benign executables.

Related work most closely related to this paper was done by Austin et al. who [5] tested four different compilers. The test data consisted of 92 separate programs: 24 were compiled with GCC, 24 with CLANG, 21 with Turbo C, and 23 with MinGW. Hidden Markov models (HMM) were then built for each program and for each compiler. Initially, the results were not very good. However, accuracy did improve to more than 90%, when the dataset was limited to the opcodes that account for at least 20% of the observations. It was unable to reliably distinguish between programs built from hand-written assembly and compiled code. The HMM did manage to accurately identify certain virus families.

N-gram analysis: N-gram scoring mechanisms were also being developed and analyzed to use with opcodes. An n-gram is a sequence of n items from a larger sequence of f.g. text or speech. In the case of opcodes, a 3-gram opcode of "a1" "b2" and "c3" would become "ab1c23". The amount of n-gram opcodes increases exponentially as n increases [6] so feature selection becomes necessary to filter out the less significant features. In the research of Santos [7], feature selection (FS) was used to reduce the training sets. Another method to reduce the training set is to use Instance Selection (IS).

The research of Kang et al. [6] focused specifically on Android Malware detection. Up to 10-gram opcodes were tested with different machine learning algorithms. The best performing algorithm was Support vector machine (SVM) when using 4-grams, which showed a 98% detection rate. Random Forest (RF) and partial decision tree (PART) came close.

Hidden Markov model (HMM): Combining opcode statistics with machine learning techniques proved to be quite effective to detect metamorphic malware. Wong & Stamp [8] set the benchmark by using tools based on HMM to detect metamorphic viruses. Many variations of using HMM have been proposed since then [5], including using chi-squared in combination with HMM [9].

Graphing: Anderson et al. [10] and Runwal et al. [11] were processing the opcodes using graphing techniques. Deshpande [12] investigated algebra methods such as eigenvalue and eigenvectors to preprocess the graph for machine learning. This was to detect the highly metamorphic MWOR worm. This extended the work of Saleh et al [13].

Hashemi et al. [14] did graph normalization and graph embedding using a “power iteration” method. Machine learning classifiers k-nearest neighbors (KNN) and SVM were then used and applied. The classifier AdaBoost offered the highest accuracy (96.09%) on a dataset with 2000 samples. SVM and KNN (K=10) scored 95.62% and 95.09% respectively. On a larger dataset with 22,200 samples, SVM, Decision Tree (DT), KNN (K=1000) and AdaBoost performed the best. Their proposed method also showed advancements compared to the methods of Santos et al. [7] and Eskandari et al [15].

ML classifiers: Shabtai et al. [16] did a comprehensive test on 8 commonly used classification algorithms and settings to distinguish between malicious and benign executables. The best-averaged settings were 2-gram (up to 6-grams were tested), normalised term frequency (TF) representation and 300 top features selected by the document frequency (DF) measure. The best classifier turned out to be Random Forest (RF) with 95.146% accuracy, with Boosted Decision Tree (BDT) and Decision Tree (DT) being 2nd and 3rd respectively. Finally, a clear performance improvement was shown when one would keep the training set up to date with recent malware on a yearly basis.

Santos et al. [7] extracted the assembly code of benign and malware executables and trained machine-learning algorithms to make a distinction between the two. This was done with opcode sequences of 2, which resulted in a high accuracy. Four machine learning models were trained, each with different learning algorithms for that model: DT, KNN, Bayesian networks (BN) and SVM. SVM: Normalised Polynomial scored the best with 95.90% accuracy. DT: Random Forest N=10 and SVM: Polynomial scored 94.98% and 95.50% respectively.
Finally, Mohammad et al. [17] used feature extraction and DT learning to decide whether a binary contains malware. 6 different decision trees were constructed. The most efficient was the random forest (RF) algorithm, which resulted in zero false positives from sets of 227, 120 and 20 opcodes. The performance was also acceptable. It managed to detect all tested classes of malware (NGVCK, G2, MPCGEN, and VCL).

Others: Shanmugam et al. [18] took a slightly different approach for measuring the similarities in opcode sequences. The method used is inspired by substitution cypher analysis. The opcode sequence of a file is given a score on how close it comes statically to a certain metamorphic family. If the statistics fit with the family statistics than the file is classified as a member of that metamorphic family.

Another approach is to use the structural entropy of the binary for matching. This technique showed excellent results on MWOR Worms but was moderately successful against detecting the NGVCK virus [19].

II. METHODS

This paper primarily focused on the statistical differences in opcode frequencies for different compiler scenarios. So instead of comparing different classes of malware, different compiler settings are compared.

The statistics are done on a collection of applications. Meaning a collection of applications are compiled with a certain setting and then all opcodes of that collection are combined before statistics are done on them. This approach is chosen because:

- Not all programs have exactly the same distributions of opcodes and some programs may react differently to certain compiler settings as opposed to others. To generalise these changes between compiler environments, the applications were combined.
- This is the main approach taken in almost all of the related works. Billar [2], for example, analysed the opcode frequencies of different collections of malware versus a collection of goodware. Machine learning classifiers were also trained using large collections of malware.

First, the collection of applications had to be compiled with different GCC versions and different compiler flags. This resulted in two datasets:

- A collection of binaries that have been compiled with 6 different compiler flags.
- A collection of binaries that have been compiled with 8 different GCC versions.

The opcodes were extracted for each collection of binaries using objdump. Then the opcodes were counted for each collection of binaries.

First the frequency of individual opcodes (1-gram) and then the frequency of opcode pairs (2-gram). According to the related works, opcode pairs should show stronger variations.

The top 30 opcodes are then taken and compared with each compiled version and setting. The rest of the opcodes are grouped into "OTHER". Finally, a statistical analysis was done on the data to find out significant differences in the opcode distribution and to identify opcodes with the largest deviations.

A. Chosen applications

Commonly used applications and Linux utils were chosen for the dataset. The source code of the programs had to be primarily written in C. The dataset contains mathematical software (gap), web services, crypto software, and hashing software, common system utilities and other common software. This in an attempt to create a balanced dataset.

The following contains the list of all programs that have been compiled. Some programs could not compile with certain optimisation flags and are not included in the dataset for the different compiler flags (*):

- barcode - part of barcode-0.99
- bash - part of bash-4.4
- cp - part of coreutils-8.28
- enscript - part of enscript-1.6.6
- find - part of findutils-4.6.0
- gap* - part of gap-4.8.9
- gcal2txt - part of gcal-4
- gcal - part of gcal-4
- git-shell - part of git 2.7.4
- git - part of git 2.7.4
- lighttpd - part of lighttpd-1.4.48
- locate - part of findutils-4.6.0
- ls - part of coreutils-8.28
- mv - part of coreutils-8.28
- openssl* - part of openssl-1.0.2n
- postgresql* - part of postgresql-10.1
- sha256sum - part of coreutils-8.28
- sha384sum - part of coreutils-8.28
- units - part of units-2.16
- vim - part of vim version 8.0.1391

Some of the binaries take up more space then others, which can be seen in the pie-chart in Figure 1. However, the dataset still remains relatively balanced:
Figure 1: Pie chart that represents the sizes of the compiled binaries.

All of the binaries have been compiled from the same machine, running Ubuntu 16.04.3 LTS. The executable file format for all binaries in the datasets are 64-bit ELF (Executable and Linking Format), which is one of the standard binary formats of Unix-like systems.

B. Compiler versions

The following versions of GCC were used to compile the programs:
- GCC: (Ubuntu/Linaro 4.4.7-8ubuntu7) 4.4.7
- GCC: (Ubuntu/Linaro 4.6.4-6ubuntu6) 4.6.4
- GCC: (Ubuntu/Linaro 4.7.4-3ubuntu12) 4.7.4
- GCC: (Ubuntu 4.8.5-4ubuntu2) 4.8.5
- GCC: (Ubuntu 4.9.4-2ubuntu16.04) 4.9.4
- GCC: (Ubuntu 5.4.1-2ubuntu16.04) 5.4.1 20160904
- GCC: (Ubuntu/Linaro 6.3.0-18ubuntu2 16.04) 6.3.0 20170519
- GCC: (Ubuntu 7.2.0-1ubuntu16.04) 7.2.0

The GCC version can be selected by supplying the `CC=` flag to the shell. No other parameters were supplied to the compiler other than the parameters that are already in the makefile of the program.

Strip

Binaries found in `/usr/bin/` and retrieved from repositories are often stripped. Stripping is a common practice where strings and comments are removed from the binary to save space. Stripping away comments and strings should not affect the number of opcodes in a binary. This was confirmed with the binaries `git`, `sha256sum` and `ls`. Therefore all binaries have been analysed unstripped for this experiment.

C. Compiler optimisation flags

The GCC optimization flags can be selected by supplying the `CLAGS==` flag to the shell. GCC: (Ubuntu 5.4.1-2ubuntu16.04) 5.4.1 20160904 was the compiler version used to compile the binaries for this dataset.

- **-O0**
  Default optimisation of GCC. [20]

- **-O1**
  Light optimization. This optimizes the binary without significantly increasing the compilation time. This acts as a macro for numerous optimizations that can be also be defined separately.

- **-O2**
  Increased optimization. This enables all optimizations that don’t come with a space trade-off. All optimisation flags of `-O1` are enabled along with additional flags.

- **-O3**
  Turns on all optimizations of `-O2`, along with additional flags. Compilation using this flag should take longer to complete.

- **-Os**
  A flag to optimize a binary for size. This enables all the `-O2` optimizations along with other flags that reduce the size.

- **-Ofast**
  Optimize for speed. This enables all `-O3` optimizations along with other (non-standardized) flags such as `-fast-math`. Some programs refuse to compile with this optimization such as OpenSSL.

D. Statistical Analysis:

Each collection of binaries has been analyzed by the individual (1-gram) and opcode pair (2-grams) statistics. The statistical analysis has been applied to the absolute number of opcodes. The absolute numbers can be found in the Appendix.

Relative frequencies

The results are presented in relative frequencies (in percentages %). Tables with the absolute number of opcodes can be found in the Appendixes.

The differences in relative frequencies have also been added to the tables. This has been calculated by subtracting the smallest relative frequency from the largest.

Z-scores

The Z score indicates the number of standard deviations an observation deviates from the mean. This will give insight into what opcode went through the strongest change at a certain setting. At the same time, it is easier to observe what opcode increased or decreased by value. The Z-score is calculated for each cell as such [21]

\[
Z = \frac{X - \mu}{\sigma}
\]

where \(X\) is the value of the cell, \(\mu\) is the mean of the row and \(\sigma\) is the standard deviation of the row. The more the Z-score of a cell has moved away from 0 (either positive or negatively), the more the value has deviated from the mean. Note that the Z-scores have been applied per row.

Chi-squared test
The Chi-squared test is a statistical test that can be applied on matrices. It works by comparing the measured (or sampled) data with the expected data. In the case of this experiment, the expected amount of opcode values are calculated cell by cell. The expected value of a cell is an average number that is calculated by multiplying the total of the cell’s entire row and column and then dividing it by the total sum of the entire table. The formula used to calculate the expected values of the cells is:

$$E_{i,j} = \frac{R_i \cdot C_j}{N}$$

(2)

where $R_i$ is the total of the cell’s entire row, $C_j$ the total of the cell’s entire column and $N$ the total sum of the entire table. All of the expected cells will then be compared with the real measured values. The chi-squared number is then calculated as such:

$$x^2 = \sum_{i,j} \frac{(O_{i,j} - E_{i,j})^2}{E_{i,j}}$$

(3)

where $O_{i,j}$ is the cell’s real value and $E_{i,j}$ is the cell’s expected value. This formula returns the chi-squared value. The higher the chi-squared value, the more significant the differences. Using the chi-squared distribution table, a probability value ($p$) can be determined. This will test the probability that the null hypothesis is true. The null hypothesis means that there are no statistically significant differences between the measured and expected data. For example, a placebo medicine would likely confirm the null hypothesis in that the symptoms of the patients do not change compared to that of the untreated patients.

A low probability value such as $<0.05$ indicates that results differ from the expected data. In the case of the opcodes, such probability would indicate that they are not uniform, meaning, some opcode quantities are relatively larger than in comparison with other opcodes.

The chi-squared calculation has been done on a matrix containing the top 30 opcodes. The remaining opcodes that were listed under “OTHER” have not been included in this score. It has to be noted that for all tables in this paper, $p = 0$. This means that there is a near 100% probability that significant differences will be found between the compiler settings/versions. However, it is hard to tell if these differences are meaningful due to the large sample size. So a way to measure the differences between opcodes regardless of sample size (effect size) is needed. This is done using Cramér’s V.

Cramér’s V

The Cramér’s V is a measure of association, which is based on the chi-squared statistic. The Cramér’s V can be used to determine differences in data on a scale between 0 and 1 that indicates the strength of a relationship. The Cramér’s V is calculated as such [23]

$$V = \sqrt{\frac{x^2}{n \cdot \min(r - 1, c - 1)}}$$

(4)

where $x^2$ is the chi-squared value, $n$ the total sum of the entire table, $r$ is the number of rows and $c$ is the number of columns. This returns a number between 1 and 0. The following guidelines are used to interpret the Cramér’s V numbers [24]:

- $<0.10$ indicates a weak relationship between the variables
- $0.10 - 0.30$ indicates a moderate relationship
- $>0.30$ indicates a strong relationship

III. RESULTS

The results of the experiments are laid-out in this section. The implications of these results will be discussed in the Discussion section.

A. GCC versions (1-gram)

In Figure 2 the relative frequencies of the opcodes are shown for each version of GCC along with a bar chart of the differences in relative frequencies. The mov opcode is by far the most common opcode, followed by callq, test and je. These four opcodes comprise 50% of all opcodes. The bar chart in this figure show that these opcodes do not show the largest variation in relative sizes. The opcodes with the greatest variances were push, pop, nop and movl.

Figure 3 shows the Z-scores and the 2 greatest deviators after counting the opcode pairs. When looking at push and pop opcodes more closely we can see that the number of opcodes significantly increase at GCC 4.8, leading to a difference of almost 50%. The Z-scores also show that the top 15 opcodes generally increase in opcode size with newer GCC versions, except for the mov opcode. Most of the large opcode deviations can be found in the older GCC versions.

Finally, the negative and positive Z-scores appear to be for the largest part clustered together. Meaning that a pattern of negative/positive z-scores is followed by a pattern of positive/negative z-scores. This shows that the opcode distribution is not random.

B. GCC versions (2-gram)

The opcode pairs (2-gram) of the binaries have also been analysed. Figure 4 shows the relative frequencies along with a bar chart. Figure 5 shows the Z-scores and the 2 greatest deviators after counting the opcode pairs.

Again, the mov and callq opcodes contribute the most to the total amount of opcodes. mov,mov is the most common opcode pair, followed by mov,callq , callq,mov and mov,xor.
The relative frequencies of the opcodes for each GCC version. The cells have been coloured based on size for each row. Green indicates the largest value and red versa. Above the table are the results of the statistical analysis. The leftmost column holds the total average for each row. The rightmost column holds the differences in relative frequencies, which has been calculated by subtracting the smallest relative frequency from the largest. The bar-chart on the right gives a visual representation of the differences in relative frequency.

The Z-scores of the opcodes for each GCC version. The cells have been coloured based on size. This has been done for the entire table to put more emphasis on the exceptional Z-scores. The stronger the colour, the greater the Z-score and therefore the greater the opcode has deviated from its mean. The two bargraphs on the right represent the two opcodes that show the largest deviations overall. It shows the absolute number of opcodes between the compiler versions.
Figure 4: Relative frequencies of opcodes for different GCC versions (2-gram).

<table>
<thead>
<tr>
<th>Opcode</th>
<th>GCC 4.6</th>
<th>GCC 4.6</th>
<th>GCC 4.7</th>
<th>GCC 4.8</th>
<th>GCC 4.9</th>
<th>GCC 5</th>
<th>GCC 6</th>
<th>GCC 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov, mov</td>
<td>0.34</td>
<td>0.36</td>
<td>0.42</td>
<td>0.46</td>
<td>0.48</td>
<td>0.54</td>
<td>0.57</td>
<td>0.66</td>
</tr>
<tr>
<td>mov, cal</td>
<td>1.21</td>
<td>1.22</td>
<td>1.24</td>
<td>1.26</td>
<td>1.27</td>
<td>1.30</td>
<td>1.32</td>
<td>1.35</td>
</tr>
<tr>
<td>mov, sar</td>
<td>-1.02</td>
<td>-1.03</td>
<td>-1.05</td>
<td>-1.07</td>
<td>-1.09</td>
<td>-1.12</td>
<td>-1.15</td>
<td>-1.18</td>
</tr>
<tr>
<td>mov, lsr</td>
<td>-2.24</td>
<td>-2.26</td>
<td>-2.28</td>
<td>-2.30</td>
<td>-2.32</td>
<td>-2.34</td>
<td>-2.36</td>
<td>-2.38</td>
</tr>
<tr>
<td>xor, mov</td>
<td>0.41</td>
<td>0.43</td>
<td>0.45</td>
<td>0.47</td>
<td>0.49</td>
<td>0.51</td>
<td>0.53</td>
<td>0.55</td>
</tr>
<tr>
<td>xor, pop</td>
<td>-0.52</td>
<td>-0.53</td>
<td>-0.54</td>
<td>-0.55</td>
<td>-0.56</td>
<td>-0.57</td>
<td>-0.58</td>
<td>-0.59</td>
</tr>
</tbody>
</table>

Figure 5: Z-scores and the 2 greatest deviators for different GCC versions (2-gram).
The barchart also shows that again push, push and pop, pop bring the largest deviations. The 2-gram Z-scores also looks somewhat similar to that of the 1-gram Z-scores. For example, the cmpb opcode in figure 3 under GCC 4.4 shows a large negative Z-score, which also holds true for the cmpb-je combination in figure 5.

When comparing the differences in relative frequencies between 1-gram and 2-grams, there seem to be larger variations with 2-grams compared to 1-grams. push, push deviates more strongly (0.61) then the 1-gram push (0.49) and other opcodes also show greater differences in relative frequencies. Also, the Cramer’s V statistic is slightly higher (0.037 vs 0.026), which indicates that there is a larger relationship between frequency and GCC version, even though overall it remains weak.

The Cramer’s V of both the 1-gram and 2-gram tables are >0.10, which indicates a moderate relationship between opcode count and the optimisation flags.

C. Flags (1-gram)

Figure 6 and 7 show the results of the analysis of opcode frequencies when compiling with different optimisation flags. By looking at the Z-table for binaries that are not optimised (flag -O0) it can be seen quite clearly that the main optimisation lies with the mov opcode. Without optimisation, the mov takes 50% of instructions. After optimisation this is reduced to around 33%. Other opcodes do increase in number but in absolute numbers, this is less than what has been saved in the number of mov opcodes (2093283 vs 1336385). The absolute numbers can be found in the Appendix.

As expected, the differences in relative frequencies for optimisation flags are much larger that of the GCC version comparison. This is also reflected in the Cramers’ V, which is 0.136. A Cramer’s >0.10 indicates that there is a moderate relationship between the number of opcodes and the optimization flag used.

The greatest deviators were nopl, nopw, cmpb and pop. By looking at the 2 greatest deviators (nopl and nopw) we see a large difference between 0,1,s and 2,3,fast.

In the Z-table, the -Os flag (size optimisation) opcodes stand out the most. Most of the opcodes deviate negatively from the mean, with the exception of the or opcodes.

D. Flags (2-gram)

Figure 8 and 9 show the 2-gram analysis for the flags. The differences between 1-gram and 2-gram are similar to what has been observed with the GCC version dataset. The differences in relative frequencies for 2-grams are larger compared to that of the 1-gram opcodes frequencies. This is reflected in the Cramers’ V, which for the 2-gram is slightly higher than that of the 1-gram table. The Z-scores also look similar to that of the Z-scores of the 1-gram. E.g. the 1-gram pop and push and the 2-gram pop, pop and push, push both show strong deviations when the -O0 flag is used.

IV. DISCUSSION

In the related works section, we saw that opcode frequencies can be used to detect if a binary belongs to a certain malware class. The goal of this paper was to determine if there is potential to apply the same techniques for different GCC versions and optimisation flags.

The frequency tables and the Z-square tables show visible patterns in the opcode frequencies. In other words, some opcodes deviate more strongly than others, which can serve as weights for machine learning training. I think this shows that using opcode frequencies has potential to detect different GCC versions and flags. However, in the case of the GCC versions, this will likely be more difficult. The Cramer’s V for the version matrices are poor which means that there is a weak relationship between opcode frequency and GCC version. Meaning the changes between GCC versions are not so clear-cut and it remains to be seen if a machine learning classifier can be accurate in differentiating between GCC versions when supplied with a binary.

The Flag matrices, on the other hand, show a moderate Cramer’s V score. Meaning that detecting optimization flags will be much more likely. But will this be enough to successfully train a classifier? Opcodes can be used to reliably identify certain virus families, but in the related work of Austin et al. [5], opcodes were unable to reliably distinguish between programs built from hand-written assembly and compiled code.

In the related work on N-gram analysis by Kang et al. [6] it was already pointed out that n-grams larger than 1 perform better than 1-grams. This is also reflected in this research. There is a higher Cramer’s V score for the 2-gram matrixes (table I) compared to the 1-gram matrixes in the results. This means that there is a stronger relationship visible and so this would provide stronger detectable variations. This, in turn, will improve the accuracy of the classifier.

<table>
<thead>
<tr>
<th>Dataset (GCC 5)</th>
<th>Chi-squared</th>
<th>p</th>
<th>Cramers V</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-gram</td>
<td>184522.4</td>
<td>0.0055</td>
<td></td>
</tr>
<tr>
<td>2-gram</td>
<td>146756.3</td>
<td>0.037</td>
<td></td>
</tr>
<tr>
<td>Flags 1-gram</td>
<td>668066.8</td>
<td>0.116</td>
<td></td>
</tr>
<tr>
<td>Flags 2-gram</td>
<td>370972.1</td>
<td>0.136</td>
<td></td>
</tr>
</tbody>
</table>

Table I: Analysis of matrixes

Improvement to this research would be the dataset. The opcode contributions per application could have been more evenly distributed. The pie-chart in figure 1 shows that 5 programs are responsible for 79% of all the opcodes and this may have degraded the statistics. The results would have been better if most applications provide an equal share of opcodes. Still, this doesn’t take away from
Figure 6: Relative frequencies of opcodes for different Flags (1-gram).

Figure 7: Z-scores and the 2 greatest deviators for different Flags (1-gram).
Figure 8: Relative frequencies of opcodes for different Flags (2-gram).

Figure 9: Z-scores and the 2 greatest deviators for different Flags (2-gram).
the fact that changing GCC settings have an effect on the opcode distributions and frequencies. And that this creates an avenue for future research for applying machine learning to detect compiler environments.

V. Conclusion

The opcode frequency distributions of binaries were measured that were compiled with different compiler versions and optimisation flags. The Z-scores were measured as well as the Cramers V. Also the differences between 1-gram and 2-gram opcodes were measured.

The 2-gram opcodes (opcode pairs) show a slightly stronger relationship then compared to single opcodes. This confirms related work about n-grams.

Statistically, the relationships between opcode and GCC versions are weak. The relationships between opcodes and optimisation flags are moderate. But however weak, patterns are visible and detectable differences do occur. Looking at the success of detecting metamorphic software using opcode frequencies, I believe that there is at least ground for further research. By seeing if a machine learning can be applied to detect compiler versions and/or compiler flags. But this may only happen if a dataset large enough can be created.

A. Future work

The challenge currently lies with the creation of the dataset. There is plenty of related work for applying machine learning on opcodes, but this requires a decent dataset. There are large malware collections available, f.g. the VX Heaven collection [25]. However, such collections for different optimisations or GCC versions do not exist yet. For this paper, the dataset was created manually, which was quite labour intensive. Having an environment that can automate this for a large set (around 200) applications would be very useful, if not mandatory to train an accurate classifier. Making use of existing reproducible build or compiler flags. This to determine whether changes in the environment would affect applications in the same manner.

VI. Acknowledgements

I would like to thank Armijn Hemel from Tjaldur Governance Solutions for supervising this research project and providing helpful feedback. Furthermore, I thank my fellow students of OS3 for the moral support and helpful discussions while working on this research project.

References


VII. APPENDIX

The appendix contains the following items:
1) Figure 1: Relative frequencies of opcodes between the individual applications of the dataset.
2) Figure 2: Absolute number of opcodes between the individual applications of the dataset.
3) Figure 3: Absolute number of opcodes for different GCC versions (1-gram).
4) Figure 4: Absolute number of opcodes for different GCC versions (2-gram).
5) Figure 5: Absolute number of opcodes for different optimization flags (1-gram).
6) Figure 6: Absolute number of opcodes for different optimization flags (2-gram).
The relative frequencies of the opcodes for each application of the dataset. The applications in this table are compiled using GCC 5 with no additional flags. The cells have been coloured based on size for each row. Green indicates the largest value and red visa versa. Above the table are the results of the statistical analysis. The leftmost column holds the total average for each row.

The absolute values of the opcodes for each application are shown in Figure 2. The two columns hold the mean and the standard deviations, which are used for calculating the Z-scores.
### Figure 3: Absolute number of opcodes for different GCC versions (1-gram).

The absolute values of the opcodes for different compiler versions. The rightmost two columns hold the mean and the standard deviations, which are used for calculating the Z-scores.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Total</th>
<th>GCC 4.4</th>
<th>GCC 4.5</th>
<th>GCC 4.6</th>
<th>GCC 4.7</th>
<th>GCC 4.8</th>
<th>GCC 4.9</th>
<th>GCC 5</th>
<th>GCC 6</th>
<th>GCC 6.1</th>
<th>GCC 7</th>
<th>μMean</th>
<th>σ or StDev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTL</td>
<td>2744775</td>
<td>306024</td>
<td>306526</td>
<td>305131</td>
<td>311439</td>
<td>389798</td>
<td>388682</td>
<td>387586</td>
<td>385856</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>934897</td>
<td>1074312</td>
<td>1079552</td>
<td>1124682</td>
<td>1190775</td>
<td>1179767</td>
<td>1180245</td>
<td>1183399</td>
<td>1187674</td>
<td>1198631</td>
<td>1205905</td>
<td>57951</td>
<td></td>
</tr>
<tr>
<td>stdv</td>
<td>1284166</td>
<td>138491</td>
<td>146789</td>
<td>150472</td>
<td>157972</td>
<td>159033</td>
<td>159204</td>
<td>159362</td>
<td>159805</td>
<td>160313</td>
<td>160985</td>
<td>57561</td>
<td></td>
</tr>
<tr>
<td>sir</td>
<td>5284</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>57561</td>
</tr>
<tr>
<td>npar</td>
<td>217248</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
</tr>
<tr>
<td>jcl</td>
<td>821553</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
</tr>
<tr>
<td>μ</td>
<td>1284166</td>
<td>138491</td>
<td>146789</td>
<td>150472</td>
<td>157972</td>
<td>159033</td>
<td>159204</td>
<td>159362</td>
<td>159805</td>
<td>160313</td>
<td>160985</td>
<td>57561</td>
<td></td>
</tr>
<tr>
<td>σ</td>
<td>5284</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>57561</td>
<td></td>
</tr>
<tr>
<td>μMean</td>
<td>934897</td>
<td>1074312</td>
<td>1079552</td>
<td>1124682</td>
<td>1190775</td>
<td>1179767</td>
<td>1180245</td>
<td>1183399</td>
<td>1187674</td>
<td>1198631</td>
<td>1205905</td>
<td>57951</td>
<td></td>
</tr>
<tr>
<td>σ or StDev.</td>
<td>1284166</td>
<td>138491</td>
<td>146789</td>
<td>150472</td>
<td>157972</td>
<td>159033</td>
<td>159204</td>
<td>159362</td>
<td>159805</td>
<td>160313</td>
<td>160985</td>
<td>57561</td>
<td></td>
</tr>
<tr>
<td>pos</td>
<td>6773</td>
<td>6773</td>
<td>6773</td>
<td>6773</td>
<td>6773</td>
<td>6773</td>
<td>6773</td>
<td>6773</td>
<td>6773</td>
<td>6773</td>
<td>6773</td>
<td>6773</td>
<td>6773</td>
</tr>
<tr>
<td>neg</td>
<td>15183</td>
<td>15183</td>
<td>15183</td>
<td>15183</td>
<td>15183</td>
<td>15183</td>
<td>15183</td>
<td>15183</td>
<td>15183</td>
<td>15183</td>
<td>15183</td>
<td>15183</td>
<td>15183</td>
</tr>
<tr>
<td>No.</td>
<td>45538</td>
<td>45538</td>
<td>45538</td>
<td>45538</td>
<td>45538</td>
<td>45538</td>
<td>45538</td>
<td>45538</td>
<td>45538</td>
<td>45538</td>
<td>45538</td>
<td>45538</td>
<td>45538</td>
</tr>
<tr>
<td>μMean</td>
<td>934897</td>
<td>1074312</td>
<td>1079552</td>
<td>1124682</td>
<td>1190775</td>
<td>1179767</td>
<td>1180245</td>
<td>1183399</td>
<td>1187674</td>
<td>1198631</td>
<td>1205905</td>
<td>57951</td>
<td></td>
</tr>
<tr>
<td>σ or StDev.</td>
<td>1284166</td>
<td>138491</td>
<td>146789</td>
<td>150472</td>
<td>157972</td>
<td>159033</td>
<td>159204</td>
<td>159362</td>
<td>159805</td>
<td>160313</td>
<td>160985</td>
<td>57561</td>
<td></td>
</tr>
</tbody>
</table>

The absolute values of the opcodes for different compiler versions. The rightmost two columns hold the mean and the standard deviations, which are used for calculating the Z-scores.

### Figure 4: Absolute number of opcodes for different GCC versions (2-gram).

The absolute values of the 2-gram opcodes for different compiler versions. The rightmost two columns hold the mean and the standard deviations, which are used for calculating the Z-scores.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Total</th>
<th>GCC 4.4</th>
<th>GCC 4.5</th>
<th>GCC 4.6</th>
<th>GCC 4.7</th>
<th>GCC 4.8</th>
<th>GCC 4.9</th>
<th>GCC 5</th>
<th>GCC 6</th>
<th>GCC 6.1</th>
<th>GCC 7</th>
<th>μMean</th>
<th>σ or StDev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTL</td>
<td>2744775</td>
<td>306024</td>
<td>306526</td>
<td>305131</td>
<td>311439</td>
<td>389798</td>
<td>388682</td>
<td>387586</td>
<td>385856</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>934897</td>
<td>1074312</td>
<td>1079552</td>
<td>1124682</td>
<td>1190775</td>
<td>1179767</td>
<td>1180245</td>
<td>1183399</td>
<td>1187674</td>
<td>1198631</td>
<td>1205905</td>
<td>57951</td>
<td></td>
</tr>
<tr>
<td>stdv</td>
<td>1284166</td>
<td>138491</td>
<td>146789</td>
<td>150472</td>
<td>157972</td>
<td>159033</td>
<td>159204</td>
<td>159362</td>
<td>159805</td>
<td>160313</td>
<td>160985</td>
<td>57561</td>
<td></td>
</tr>
<tr>
<td>sir</td>
<td>5284</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>57561</td>
<td></td>
</tr>
<tr>
<td>npar</td>
<td>217248</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
<td>218126</td>
</tr>
<tr>
<td>jcl</td>
<td>821553</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
<td>874949</td>
</tr>
<tr>
<td>μ</td>
<td>1284166</td>
<td>138491</td>
<td>146789</td>
<td>150472</td>
<td>157972</td>
<td>159033</td>
<td>159204</td>
<td>159362</td>
<td>159805</td>
<td>160313</td>
<td>160985</td>
<td>57561</td>
<td></td>
</tr>
<tr>
<td>σ</td>
<td>5284</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>4892</td>
<td>57561</td>
<td></td>
</tr>
<tr>
<td>μMean</td>
<td>934897</td>
<td>1074312</td>
<td>1079552</td>
<td>1124682</td>
<td>1190775</td>
<td>1179767</td>
<td>1180245</td>
<td>1183399</td>
<td>1187674</td>
<td>1198631</td>
<td>1205905</td>
<td>57951</td>
<td></td>
</tr>
<tr>
<td>σ or StDev.</td>
<td>1284166</td>
<td>138491</td>
<td>146789</td>
<td>150472</td>
<td>157972</td>
<td>159033</td>
<td>159204</td>
<td>159362</td>
<td>159805</td>
<td>160313</td>
<td>160985</td>
<td>57561</td>
<td></td>
</tr>
</tbody>
</table>

The absolute values of the 2-gram opcodes for different compiler versions. The rightmost two columns hold the mean and the standard deviations, which are used for calculating the Z-scores.
Figure 5: Absolute number of opcodes for different optimization flags (1-gram).

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Total</th>
<th>O</th>
<th>1</th>
<th>2</th>
<th>s</th>
<th>text</th>
<th>μ</th>
<th>Mean</th>
<th>o</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>3519497</td>
<td>109438</td>
<td>109592</td>
<td>109647</td>
<td>204629</td>
<td>395385</td>
<td>204185</td>
<td>177085</td>
<td>129355</td>
<td></td>
</tr>
<tr>
<td>mov,R</td>
<td>390349</td>
<td>1040046</td>
<td>562151</td>
<td>562510</td>
<td>675712</td>
<td>436980</td>
<td>676974</td>
<td>633090</td>
<td>134515</td>
<td></td>
</tr>
<tr>
<td>mov,call</td>
<td>4889</td>
<td>111841</td>
<td>139154</td>
<td>111313</td>
<td>163906</td>
<td>125294</td>
<td>162996</td>
<td>168716</td>
<td>16830</td>
<td></td>
</tr>
<tr>
<td>je</td>
<td>129866</td>
<td>74237</td>
<td>74082</td>
<td>79616</td>
<td>113760</td>
<td>80359</td>
<td>112414</td>
<td>80596</td>
<td>153881</td>
<td></td>
</tr>
<tr>
<td>test</td>
<td>513724</td>
<td>67138</td>
<td>67058</td>
<td>85689</td>
<td>116881</td>
<td>60922</td>
<td>116156</td>
<td>84521</td>
<td>25247</td>
<td></td>
</tr>
<tr>
<td>cmp</td>
<td>50087</td>
<td>45940</td>
<td>39759</td>
<td>56137</td>
<td>60526</td>
<td>46227</td>
<td>81034</td>
<td>92976</td>
<td>16133</td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>55216</td>
<td>55014</td>
<td>54660</td>
<td>54712</td>
<td>71914</td>
<td>4760</td>
<td>71914</td>
<td>4760</td>
<td>10269</td>
<td></td>
</tr>
<tr>
<td>push</td>
<td>128627</td>
<td>15356</td>
<td>12327</td>
<td>69431</td>
<td>83569</td>
<td>65438</td>
<td>83616</td>
<td>54771</td>
<td>92964</td>
<td></td>
</tr>
<tr>
<td>lea</td>
<td>120331</td>
<td>41414</td>
<td>34982</td>
<td>54395</td>
<td>55351</td>
<td>54494</td>
<td>34327</td>
<td>54689</td>
<td>7134</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>259698</td>
<td>60445</td>
<td>44142</td>
<td>46992</td>
<td>57149</td>
<td>26125</td>
<td>57139</td>
<td>29621</td>
<td>13916</td>
<td></td>
</tr>
<tr>
<td>jmp</td>
<td>287368</td>
<td>33106</td>
<td>31410</td>
<td>52645</td>
<td>49319</td>
<td>28938</td>
<td>69190</td>
<td>47885</td>
<td>1484</td>
<td></td>
</tr>
<tr>
<td>jsr</td>
<td>369302</td>
<td>30883</td>
<td>25667</td>
<td>21566</td>
<td>27210</td>
<td>17510</td>
<td>27234</td>
<td>24938</td>
<td>4913</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>179050</td>
<td>39324</td>
<td>29226</td>
<td>27796</td>
<td>33762</td>
<td>13946</td>
<td>31798</td>
<td>29675</td>
<td>8739</td>
<td></td>
</tr>
<tr>
<td>jmp,E</td>
<td>179346</td>
<td>46408</td>
<td>32729</td>
<td>16065</td>
<td>19371</td>
<td>31871</td>
<td>14671</td>
<td>28950</td>
<td>13417</td>
<td></td>
</tr>
<tr>
<td>mov,call</td>
<td>45338</td>
<td>30853</td>
<td>25267</td>
<td>21566</td>
<td>27210</td>
<td>17510</td>
<td>27234</td>
<td>24938</td>
<td>4913</td>
<td></td>
</tr>
<tr>
<td>mov,call</td>
<td>121116</td>
<td>136</td>
<td>129</td>
<td>3578</td>
<td>42671</td>
<td>144</td>
<td>42675</td>
<td>10816</td>
<td>2212</td>
<td></td>
</tr>
<tr>
<td>mov,R</td>
<td>104153</td>
<td>24955</td>
<td>19598</td>
<td>17428</td>
<td>16794</td>
<td>19048</td>
<td>17438</td>
<td>18074</td>
<td>3210</td>
<td></td>
</tr>
<tr>
<td>mov,E</td>
<td>32703</td>
<td>14935</td>
<td>13148</td>
<td>11543</td>
<td>16267</td>
<td>10152</td>
<td>12925</td>
<td>13794</td>
<td>2441</td>
<td></td>
</tr>
<tr>
<td>cmp</td>
<td>37809</td>
<td>29963</td>
<td>25431</td>
<td>4272</td>
<td>5338</td>
<td>18314</td>
<td>5317</td>
<td>13902</td>
<td>8034</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9275</td>
<td>19907</td>
<td>8414</td>
<td>7592</td>
<td>10960</td>
<td>831</td>
<td>13087</td>
<td>12667</td>
<td>3930</td>
<td></td>
</tr>
<tr>
<td>push</td>
<td>70091</td>
<td>12</td>
<td>11</td>
<td>21772</td>
<td>31818</td>
<td>125</td>
<td>24110</td>
<td>17395</td>
<td>17988</td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>71598</td>
<td>11207</td>
<td>10953</td>
<td>15982</td>
<td>8693</td>
<td>11973</td>
<td>10782</td>
<td>4301</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq</td>
<td>49673</td>
<td>10072</td>
<td>7575</td>
<td>7523</td>
<td>8364</td>
<td>7373</td>
<td>5365</td>
<td>6279</td>
<td>1178</td>
<td></td>
</tr>
<tr>
<td>pushq,E</td>
<td>49025</td>
<td>18311</td>
<td>5008</td>
<td>5292</td>
<td>7210</td>
<td>5032</td>
<td>7212</td>
<td>6201</td>
<td>5015</td>
<td></td>
</tr>
<tr>
<td>mov,E</td>
<td>17977</td>
<td>7881</td>
<td>7870</td>
<td>10054</td>
<td>5032</td>
<td>12956</td>
<td>9046</td>
<td>2403</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jle</td>
<td>71785</td>
<td>6815</td>
<td>6912</td>
<td>7418</td>
<td>5641</td>
<td>5263</td>
<td>15684</td>
<td>7916</td>
<td>2085</td>
<td></td>
</tr>
<tr>
<td>shl</td>
<td>38481</td>
<td>19098</td>
<td>4710</td>
<td>4340</td>
<td>6711</td>
<td>2903</td>
<td>6719</td>
<td>6414</td>
<td>1089</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>6091</td>
<td>5125</td>
<td>1864</td>
<td>4511</td>
<td>6372</td>
<td>10139</td>
<td>6180</td>
<td>4051</td>
<td>2205</td>
<td></td>
</tr>
<tr>
<td>OTHER</td>
<td>649282</td>
<td>119739</td>
<td>94006</td>
<td>52682</td>
<td>142602</td>
<td>93895</td>
<td>142215</td>
<td>103815</td>
<td>16085</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6: Absolute number of opcodes for different optimization flags (2-gram).