Opcode statistics for detecting compiler settings

MSc Research Project (#20)

Kenneth van Rijsbergen
Master Security and Network Engineering, University of Amsterdam
kenneth.vanrijsbergen@os3.nl
February 11, 2018

***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 `"a1b2c3"`. 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:

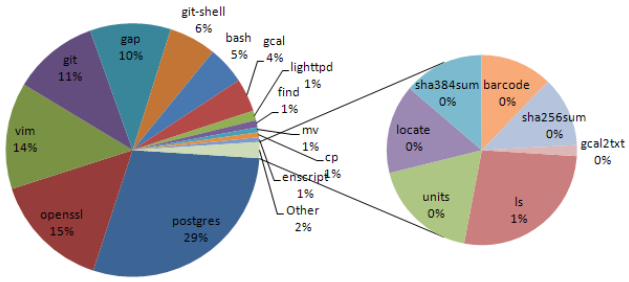


Figure 1: Pie chart that represents the sizes of the compiled binaries.

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 4.9.4-2ubuntu1 16.04) 4.9.4
- GCC: (Ubuntu 5.4.1-2ubuntu1 16.04) 5.4.1 20160904
- GCC: (Ubuntu/Linaro 6.3.0-18ubuntu2 16.04) 6.3.0 20170519
- GCC: (Ubuntu 7.2.0-1ubuntu1 16.04) 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 5.4.1-2ubuntu1 16.04) 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]

$$Z = \frac{X - \mu}{\sigma} \tag{1}$$

where X is the value of the cell, μ is the mean of the row and σ 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 [22]. The formula used to calculate the expected values of the cells is:

$$E_{i,j} = \frac{R_i \cdot C_j}{N} \tag{2}$$

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:

$$x^2 = \sum_{i,j} \frac{(O_{i,j} - E_{i,j})^2}{E_{i,j}} \tag{3}$$

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. F.g. 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 the 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 matrice 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 Cramérs V.

Cramérs V

The Cramer's V is a measure of association, which is based on the chi-squared statistic. The Cramer'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 Cramer's V is calculated as such [23]

$$V = \sqrt{\frac{x^2}{n \cdot min(r-1, c-1)}} \tag{4}$$

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 Cramer'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.

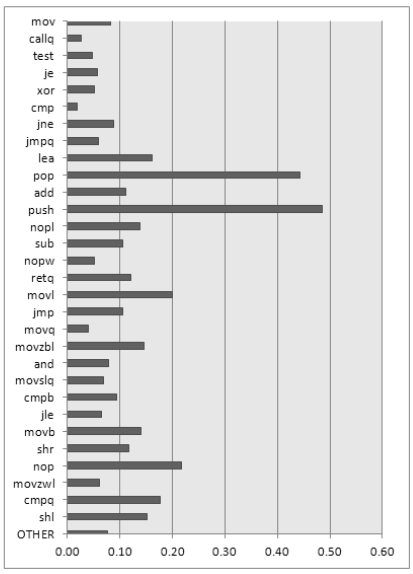
B. GCC versions (2-gram)

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`.

Figure 2: Relative frequencies of opcodes for different GCC versions (1-gram).

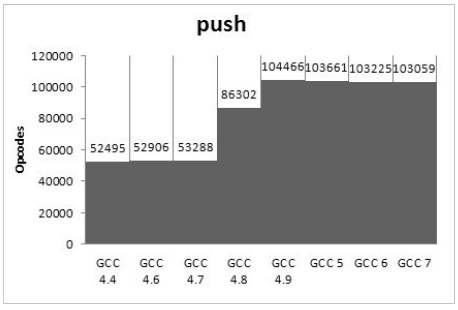
Pearson's chi-squared test (χ2)				116455.3					
Cramér's V				0.025513					
p				0					Differences in
Opcode	Average	GCC 4.4	GCC 4.6	GCC 4.7	GCC 4.8	GCC 4.9	GCC 5	GCC 6	GCC 7	relative frequencies
mov	34.90%	36.78%	36.56%	36.22%	34.27%	33.94%	33.98%	34.07%	33.70%	0.08
callq	8.30%	8.23%	8.43%	8.30%	8.21%	8.31%	8.28%	8.29%	8.30%	0.03
test	5.00%	4.94%	5.11%	5.01%	4.95%	4.86%	4.92%	4.95%	4.98%	0.05
je	4.70%	4.74%	4.81%	4.85%	4.68%	4.58%	4.66%	4.67%	4.70%	0.06
xor	4.60%	4.42%	4.66%	4.43%	4.45%	4.64%	4.63%	4.59%	4.59%	0.05
cmp	3.30%	3.33%	3.31%	3.36%	3.31%	3.29%	3.32%	3.30%	3.33%	0.02
jne	3.10%	2.93%	3.00%	3.00%	3.00%	3.15%	3.15%	3.15%	3.21%	0.09
jmpq	3.00%	3.02%	2.91%	3.10%	3.04%	3.02%	3.03%	3.03%	3.01%	0.06
lea	3.00%	2.62%	2.71%	2.87%	3.11%	3.12%	3.06%	3.06%	3.13%	0.16
pop	2.90%	2.05%	2.07%	2.10%	3.67%	3.46%	3.43%	3.34%	3.33%	0.44
add	2.90%	2.98%	3.14%	3.09%	2.87%	2.88%	2.85%	2.84%	2.79%	0.11
push	2.40%	1.55%	1.57%	1.59%	2.53%	3.01%	2.98%	2.97%	2.95%	0.49
nopl	2.30%	2.55%	2.25%	2.31%	2.33%	2.23%	2.20%	2.20%	2.31%	0.14
sub	1.50%	1.51%	1.51%	1.52%	1.41%	1.55%	1.55%	1.58%	1.58%	0.11
nopw	1.40%	1.41%	1.38%	1.44%	1.41%	1.39%	1.37%	1.36%	1.43%	0.05
retq	1.10%	1.07%	1.08%	1.12%	1.22%	1.12%	1.12%	1.10%	1.11%	0.12
movl	1.10%	1.17%	1.18%	1.18%	1.18%	1.02%	1.03%	1.03%	0.94%	0.20
jmp	1.00%	1.03%	0.92%	0.99%	0.99%	0.97%	0.98%	0.97%	0.94%	0.11
movq	1.00%	0.95%	0.97%	0.97%	0.96%	0.93%	0.95%	0.95%	0.96%	0.04
movzbl	0.90%	1.04%	0.95%	0.93%	0.93%	0.91%	0.89%	0.98%	0.91%	0.15
and	0.70%	0.65%	0.63%	0.66%	0.66%	0.67%	0.68%	0.69%	0.68%	0.08
movslq	0.60%	0.67%	0.64%	0.66%	0.67%	0.63%	0.64%	0.64%	0.63%	0.07
cmpb	0.60%	0.54%	0.60%	0.60%	0.59%	0.59%	0.58%	0.57%	0.56%	0.09
jle	0.50%	0.48%	0.48%	0.47%	0.48%	0.47%	0.47%	0.47%	0.45%	0.06
movb	0.50%	0.46%	0.48%	0.48%	0.48%	0.46%	0.46%	0.46%	0.41%	0.14
shr	0.40%	0.37%	0.39%	0.41%	0.42%	0.41%	0.42%	0.42%	0.40%	0.12
nop	0.40%	0.47%	0.41%	0.41%	0.41%	0.37%	0.37%	0.37%	0.39%	0.22
movzwl	0.40%	0.37%	0.39%	0.38%	0.38%	0.37%	0.37%	0.37%	0.37%	0.06
cmpq	0.40%	0.38%	0.38%	0.41%	0.36%	0.36%	0.36%	0.36%	0.33%	0.18
shl	0.40%	0.40%	0.38%	0.38%	0.37%	0.37%	0.35%	0.34%	0.34%	0.15
OTHER	6.90%	6.88%	6.70%	6.76%	6.67%	6.91%	6.91%	6.90%	7.23%	0.08

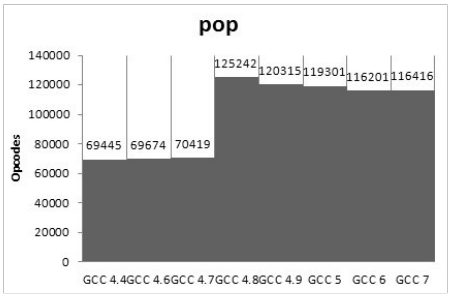


The relative frequencies of the opcodes for each GCC version. The cells have been coloured based on size for each row. Green indicates the largest value and red visa versa. Above the table are the results of the statistical analysis. The leftmost column holds the total average for each row. The rightmost 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.

Figure 3: Z-scores and the 2 greatest deviators for different GCC versions (1-gram).

Opcode	GCC 4.4	GCC 4.6	GCC 4.7	GCC 4.8	GCC 4.9	GCC 5	GCC 6	GCC 7
mov	1.74	1.08	0.58	-1	-0.65	-0.54	-0.46	-0.75
callq	-1.13	-0.25	-1.28	-0.89	0.89	0.76	0.82	1.09
test	-1.21	0.6	-0.99	-0.65	-0.55	0.34	0.79	1.65
je	-0.38	0.07	0.67	-1.12	-1.55	0.27	0.48	1.56
xor	-1.2	0.07	-1.41	-0.82	0.95	0.96	0.65	0.8
cmp	-0.49	-1.51	-0.76	-0.61	0.34	1.02	0.59	1.42
jne	-1.2	-0.91	-0.95	-0.58	0.76	0.81	0.8	1.28
jmpq	-0.44	-2.26	0.09	0.04	0.55	0.77	0.65	0.6
lea	-1.55	-1.28	-0.65	0.55	0.81	0.61	0.58	0.93
pop	-1.22	-1.21	-1.18	0.94	0.75	0.71	0.59	0.6
add	0.29	1.8	1.08	-0.88	-0.15	-0.5	-0.57	-1.07
push	-1.19	-1.17	-1.16	0.15	0.88	0.84	0.83	0.82
nopl	2.18	-0.9	-0.34	0.2	-0.4	-0.6	-0.69	0.54
sub	-0.47	-0.61	-0.61	-1.71	0.61	0.68	1	1.12
nopw	-0.14	-1.63	0.31	0.15	0.31	-0.41	-0.51	1.92
retq	-1.19	-1.18	-0.51	1.9	0.4	0.34	0.05	0.19
movl	0.83	0.85	0.74	1.08	-0.73	-0.55	-0.55	-1.66
jmp	1.17	-2.18	-0.1	0.37	0.36	0.61	0.2	-0.44
movq	-1.03	-0.68	-0.41	-0.12	-0.89	0.59	0.59	1.95
movzbl	1.97	-0.23	-0.74	-0.38	-0.37	-0.89	1.11	-0.47
and	-0.88	-1.63	-0.65	-0.28	0.62	0.93	1.11	0.77
movslq	1.49	-1.57	0.1	1.17	-0.54	0.1	0.05	-0.8
cmpb	-2.22	0.46	0.36	0.52	0.9	0.55	-0.02	-0.56
jle	1.22	-0.64	-1.28	0.49	0.21	0.91	0.54	-1.44
movb	-0.23	0.36	0.45	0.84	0.19	0.45	0.31	-2.36
shr	-1.62	-1.3	-0.31	0.76	0.63	0.92	0.87	0.06
nop	2.14	-0.04	0.16	0.45	-0.77	-0.82	-0.95	-0.18
movzwl	-1.27	1.83	-1.15	0.59	-0.34	0.47	-0.18	0.04
cmpq	0.51	0.02	1.82	-0.29	-0.07	-0.12	-0.04	-1.82
shl	1.82	0.42	0.1	-0.16	0.72	-0.66	-1.23	-1
OTHER	-0.24	-1.14	-0.98	-0.88	0.49	0.51	0.47	1.78





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).

Pearson's chi-squared test (χ2)		146756.3							
Cramér's V		0.036947							
p		0							Differences in
Opcode	Average	GCC 4.4	GCC 4.6	GCC 4.7	GCC 4.8	GCC 4.9	GCC 5	GCC 6	GCC 7	relative frequencies
mov,mov	15.50%	17.54%	17.23%	16.96%	14.84%	14.43%	14.49%	14.51%	14.15%	0.19
mov,callq	5.49%	5.56%	5.57%	5.60%	5.51%	5.45%	5.42%	5.42%	5.43%	0.03
callq,mov	3.97%	4.06%	4.12%	4.02%	4.01%	3.90%	3.89%	3.88%	3.90%	0.06
mov,xor	2.18%	2.05%	2.23%	2.03%	2.02%	2.28%	2.28%	2.28%	2.28%	0.12
test,je	2.11%	2.08%	2.13%	2.15%	2.08%	2.04%	2.12%	2.12%	2.14%	0.05
je,mov	2.09%	2.16%	2.15%	2.18%	2.07%	2.02%	2.05%	2.05%	2.04%	0.07
pop,pop	1.97%	1.35%	1.36%	1.37%	2.48%	2.34%	2.32%	2.25%	2.25%	0.46
mov,test	1.61%	1.58%	1.57%	1.60%	1.64%	1.61%	1.64%	1.65%	1.61%	0.05
callq,test	1.57%	1.59%	1.62%	1.64%	1.55%	1.54%	1.53%	1.54%	1.54%	0.07
xor,mov	1.45%	1.47%	1.42%	1.42%	1.36%	1.46%	1.46%	1.50%	1.50%	0.09
test,jne	1.28%	1.27%	1.30%	1.29%	1.28%	1.27%	1.29%	1.28%	1.25%	0.04
jne,mov	1.22%	1.20%	1.25%	1.21%	1.21%	1.21%	1.23%	1.24%	1.25%	0.04
lea,mov	1.16%	1.03%	1.09%	1.17%	1.22%	1.19%	1.19%	1.18%	1.18%	0.15
mov,jmpq	1.13%	1.09%	1.04%	1.12%	1.15%	1.15%	1.16%	1.16%	1.16%	0.10
push,push	1.12%	0.63%	0.60%	0.63%	0.93%	1.52%	1.53%	1.53%	1.53%	0.61
xor,callq	1.06%	1.02%	1.18%	1.05%	1.02%	1.06%	1.05%	1.06%	1.06%	0.14
cmp,je	1.06%	1.01%	1.08%	1.12%	1.07%	1.04%	1.06%	1.06%	1.05%	0.09
push,mov	0.95%	0.66%	0.71%	0.67%	1.18%	1.13%	1.08%	1.06%	1.06%	0.44
nopl,mov	0.92%	1.00%	0.93%	0.95%	0.88%	0.89%	0.89%	0.88%	0.95%	0.12
mov,add	0.89%	1.11%	1.10%	1.10%	0.77%	0.77%	0.78%	0.77%	0.75%	0.33
test,mov	0.87%	0.86%	0.96%	0.86%	0.86%	0.86%	0.81%	0.84%	0.88%	0.16
mov,lea	0.83%	0.72%	0.71%	0.76%	0.90%	0.89%	0.88%	0.89%	0.89%	0.21
jmpq,nopl	0.82%	0.87%	0.80%	0.82%	0.82%	0.78%	0.78%	0.78%	0.87%	0.10
jmpq,mov	0.75%	0.77%	0.71%	0.78%	0.76%	0.75%	0.77%	0.77%	0.68%	0.12
add,mov	0.69%	0.63%	0.71%	0.68%	0.74%	0.70%	0.69%	0.70%	0.68%	0.14
pop,retq	0.69%	0.52%	0.54%	0.55%	0.87%	0.77%	0.76%	0.74%	0.76%	0.41
mov,je	0.68%	0.64%	0.72%	0.68%	0.68%	0.68%	0.65%	0.67%	0.71%	0.10
sub,mov	0.65%	0.62%	0.61%	0.61%	0.52%	0.72%	0.71%	0.72%	0.72%	0.28
mov,sub	0.65%	0.72%	0.72%	0.69%	0.53%	0.65%	0.62%	0.63%	0.63%	0.26
cmp,jne	0.59%	0.54%	0.55%	0.59%	0.60%	0.59%	0.60%	0.60%	0.65%	0.16
OTHER	44.04%	43.63%	43.29%	43.70%	44.42%	44.30%	44.23%	44.24%	44.47%	0.03

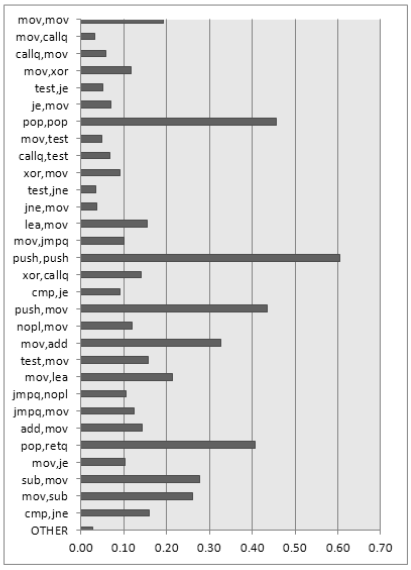
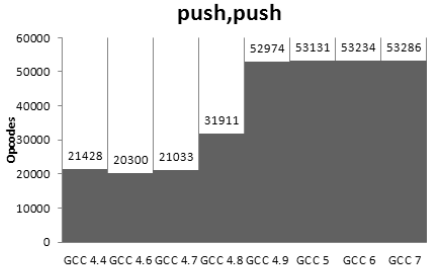
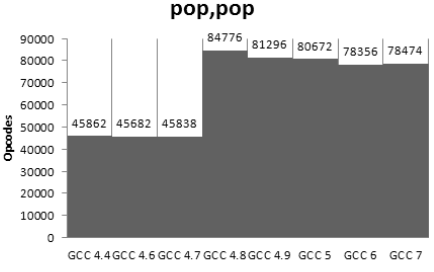


Figure 5: Z-scores and the 2 greatest deviators for different GCC versions (2-gram).

Opcode	GCC 4.4	GCC 4.6	GCC 4.7	GCC 4.8	GCC 4.9	GCC 5	GCC 6	GCC 7
mov,mov	1.52	1.14	0.89	-0.61	-0.73	-0.66	-0.65	-0.91
mov,callq	0.21	-1.73	-0.87	-0.35	1.11	0.21	0.07	1.35
callq,mov	1.21	1.52	-1.14	0.6	-0.53	-0.53	-0.98	-0.15
mov,xor	-1.02	0	-1.27	-1.16	0.85	0.84	0.85	0.91
test,je	-1.13	-0.38	-0.17	-0.82	-0.92	0.96	0.97	1.48
je,mov	1.54	0.51	1.2	-1.01	-1.25	-0.3	-0.25	-0.44
pop,pop	-1.2	-1.21	-1.2	0.94	0.75	0.72	0.59	0.6
mov,test	-0.93	-1.54	-0.94	0.4	0.4	1.04	1.14	0.43
callq,test	0.22	0.85	1.89	-1.33	-0.38	-0.71	-0.48	-0.05
xor,mov	0.01	-0.9	-0.95	-1.48	0.51	0.58	1.06	1.18
test,jne	-1.4	-0.36	-0.86	-0.33	0.48	1.64	1.06	-0.22
jne,mov	-1.2	-0.11	-1.23	-0.71	0.05	0.87	0.94	1.38
lea,mov	-1.85	-1.18	-0.2	0.74	0.7	0.64	0.52	0.64
mov,jmpq	-0.83	-1.85	-0.62	0.2	0.57	0.82	0.79	0.92
push,push	-1.05	-1.12	-1.07	-0.4	0.9	0.91	0.92	0.92
xor,callq	-1.15	1.96	-0.79	-0.98	0.19	0.17	0.23	0.37
cmp,je	-2.16	-0.03	1.24	0.04	-0.4	0.66	0.43	0.21
push,mov	-1.25	-1.08	-1.26	0.97	0.84	0.64	0.57	0.57
nopl,mov	1.8	-0.22	0.24	-1.18	-0.47	-0.52	-0.76	1.09
mov,add	1.28	1.19	1.14	-0.79	-0.71	-0.61	-0.68	-0.82
test,mov	-0.51	1.99	-0.61	-0.2	0.2	-1.27	-0.37	0.76
mov,lea	-1.23	-1.44	-0.92	0.68	0.76	0.64	0.71	0.79
jmpq,nopl	1.25	-0.78	-0.39	-0.03	-0.63	-0.54	-0.74	1.86
jmpq,mov	0.38	-1.53	0.32	0.37	0.33	0.86	0.89	-1.62
add,mov	-2.04	0.06	-0.71	1.31	0.5	0.29	0.55	0.03
pop,retq	-1.26	-1.16	-1.09	1.26	0.63	0.57	0.44	0.61
mov,je	-1.72	0.89	-0.44	-0.17	0.46	-0.6	0.02	1.56
sub,mov	-0.53	-0.67	-0.72	-1.61	0.85	0.83	0.92	0.94
mov,sub	1.2	0.91	0.44	-2.1	0.11	-0.28	-0.2	-0.07
cmp,jne	-1.39	-1.25	-0.44	0.23	0.28	0.48	0.42	1.66
OTHER	-0.8	-1.46	-1.18	0.14	0.75	0.73	0.73	1.09





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'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. F.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 betten 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.

	Chi-squared	p	Cramérs V
Dataset (GCC 5)	184522.4	0	0.055
Versions 1-gram	116455.3	0	0.025
Versions 2-gram	146756.3	0	0.037
Flags 1-gram	668066.8	0	0.116
Flags 2-gram	570972.1	0	0.136

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).

Pearson's chi-squared test (χ2)		668066.8						
Cramér's V		0.115758						
p		0						Differences in
Opcode	Average	0	1	2	3	s	fast	relative frequencies
mov	36.90%	49.92%	37.25%	32.75%	33.06%	32.72%	33.07%	0.34
callq	8.30%	7.56%	9.23%	8.25%	7.97%	9.38%	7.97%	0.19
je	4.90%	3.55%	4.93%	5.00%	5.43%	5.16%	5.44%	0.35
test	4.80%	3.22%	4.44%	5.37%	5.70%	4.56%	5.70%	0.44
cmp	3.40%	2.19%	3.69%	3.53%	3.96%	3.46%	3.96%	0.45
jne	3.30%	2.65%	3.58%	3.44%	3.52%	3.53%	3.52%	0.26
xor	3.10%	0.73%	0.82%	4.36%	4.06%	4.90%	4.06%	0.85
push	3.10%	1.97%	3.61%	3.42%	2.85%	4.60%	2.85%	0.57
lea	3.10%	2.32%	3.40%	3.11%	3.23%	3.37%	3.24%	0.32
add	2.80%	3.13%	2.86%	2.93%	2.80%	1.96%	2.80%	0.37
jmpq	2.70%	1.58%	2.26%	3.31%	3.39%	2.17%	3.38%	0.53
pop	2.50%	0.54%	2.59%	3.40%	2.84%	3.66%	2.84%	0.85
sub	1.70%	1.89%	1.93%	1.75%	1.65%	1.04%	1.65%	0.46
jmp	1.60%	2.31%	2.46%	1.06%	0.96%	2.39%	0.96%	0.61
movl	1.40%	1.47%	1.46%	1.33%	1.34%	1.32%	1.34%	0.10
nopl	1.10%	0.01%	0.01%	2.22%	2.09%	0.01%	2.09%	1.00
movzbl	1.10%	1.65%	1.06%	0.87%	1.07%	0.32%	1.07%	0.81
movq	1.00%	0.89%	1.13%	1.06%	1.06%	1.04%	1.06%	0.21
retq	1.00%	1.17%	1.12%	1.09%	0.82%	1.20%	0.82%	0.32
movslq	0.80%	0.71%	0.87%	0.72%	0.80%	0.79%	0.80%	0.18
cmpl	0.70%	1.13%	1.35%	0.27%	0.26%	1.37%	0.26%	0.81
and	0.70%	0.76%	0.56%	0.62%	0.74%	0.61%	0.74%	0.27
nopw	0.70%	0.01%	0.01%	1.37%	1.18%	0.01%	1.18%	1.00
cmpb	0.60%	0.07%	0.74%	0.67%	0.68%	0.65%	0.68%	0.90
pushq	0.50%	0.50%	0.50%	0.47%	0.41%	0.55%	0.41%	0.26
cmpq	0.50%	0.87%	0.38%	0.33%	0.35%	0.42%	0.35%	0.62
movb	0.50%	0.28%	0.52%	0.48%	0.53%	0.40%	0.53%	0.48
jle	0.40%	0.33%	0.46%	0.47%	0.51%	0.41%	0.51%	0.36
shl	0.40%	0.63%	0.31%	0.27%	0.33%	0.22%	0.33%	0.65
or	0.30%	0.24%	0.26%	0.28%	0.30%	0.76%	0.30%	0.68
OTHER	6.10%	5.72%	6.23%	5.83%	6.09%	7.03%	6.08%	0.19

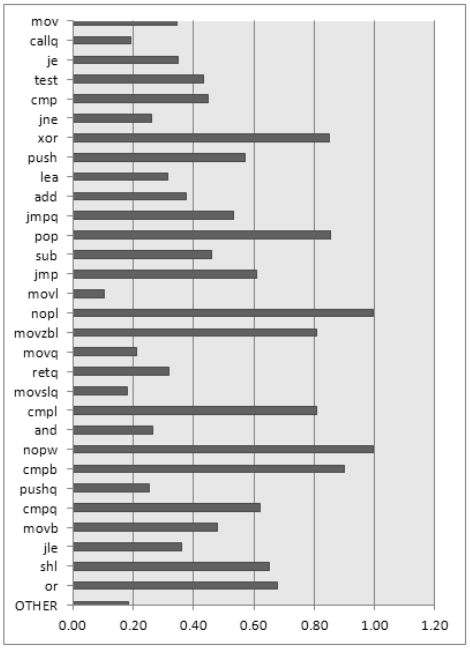
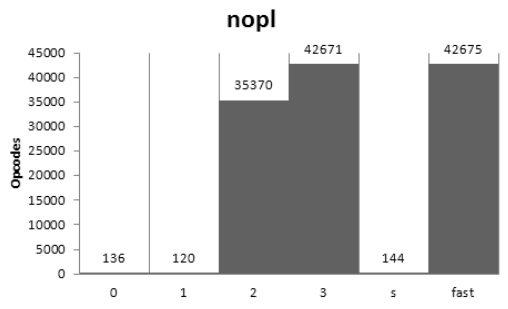


Figure 7: Z-scores and the 2 greatest deviators for different Flags (1-gram).

Opcode	0	1	2	3	s	fast
mov	1.84	-0.42	-0.62	0.11	-1.01	0.11
callq	0.69	-0.43	-0.91	0.97	-1.28	0.97
je	-0.64	-0.62	-0.36	1.27	-0.91	1.27
test	-0.72	-0.73	-0.01	1.22	-0.98	1.22
cmp	-0.94	-0.32	-0.3	1.24	-0.91	1.24
jne	-0.37	-0.5	-0.44	1.24	-1.17	1.24
xor	-1.21	-1.31	0.45	0.87	0.33	0.87
push	-1.89	-0.03	-0.04	0.51	0.95	0.51
lea	-0.63	-0.34	-0.54	1.25	-1.02	1.26
add	1.16	-0.44	-0.19	0.57	-1.66	0.57
jmpq	-0.8	-0.75	0.26	1.16	-1.03	1.15
pop	-1.87	-0.33	0.51	0.74	0.22	0.73
sub	1.13	-0.05	-0.22	0.47	-1.8	0.47
jmp	1.56	0.67	-0.97	-0.74	0.23	-0.75
movl	1.32	-0.48	-0.66	0.6	-1.38	0.6
nopl	-0.91	-0.91	0.69	1.02	-0.91	1.02
movzbl	1.56	-0.27	-0.49	0.31	-1.43	0.32
movq	0.09	-0.43	-0.47	1.12	-1.44	1.12
retq	2.02	-0.35	-0.2	-0.42	-0.63	-0.41
movslq	0.47	-0.26	-0.92	1.02	-1.34	1.02
cmpl	1.22	0.85	-0.98	-0.85	0.61	-0.85
and	1.05	-1.01	-0.59	0.82	-1.08	0.82
nopw	-0.91	-0.91	0.79	0.97	-0.91	0.97
cmpb	-1.83	0.26	0.15	0.86	-0.29	0.86
pushq	1.89	-0.6	-0.64	0.07	-0.8	0.07
cmpq	2.02	-0.48	-0.58	-0.22	-0.53	-0.22
movb	-0.94	-0.1	-0.19	1.17	-1.13	1.19
jle	-0.52	-0.48	-0.24	1.22	-1.2	1.22
shl	1.86	-0.47	-0.58	0.08	-0.98	0.09
or	-0.4	-0.93	-0.68	0.07	1.87	0.07
OTHER	0.72	-0.88	-0.95	1.01	-0.89	0.99



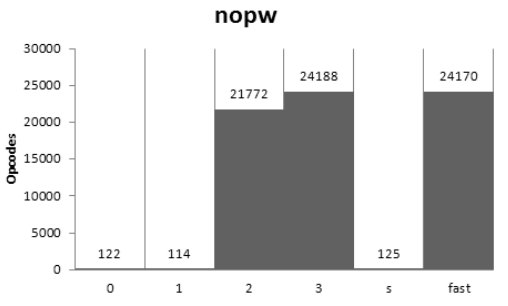


Figure 8: Relative frequencies of opcodes for different Flags (2-gram).

Pearson's chi-squared test (χ2)		570972.1						
Cramér's V		0.13632						
p		0						Differences in
Opcode	Average	0	1	2	3	s	fast	relative frequencies
mov,mov	36.90%	27.90%	18.26%	13.04%	13.22%	12.51%	13.23%	0.55
mov,callq	8.30%	6.72%	7.30%	5.17%	5.05%	5.79%	5.05%	0.31
callq,mov	4.90%	4.09%	4.71%	3.62%	3.58%	3.78%	3.58%	0.24
je,mov	4.80%	2.33%	2.13%	2.14%	2.38%	1.96%	2.37%	0.18
test,je	3.40%	1.74%	2.38%	2.29%	2.50%	2.02%	2.50%	0.30
mov,test	3.30%	1.39%	1.79%	1.96%	2.23%	0.98%	2.23%	0.56
pop,pop	3.10%	0.20%	1.84%	2.29%	2.00%	2.42%	2.00%	0.92
mov,xor	3.10%	0.60%	0.63%	2.33%	2.16%	2.58%	2.16%	0.77
jne,mov	3.10%	1.90%	1.55%	1.31%	1.40%	1.42%	1.40%	0.31
callq,test	2.80%	0.99%	1.30%	1.61%	1.56%	2.06%	1.56%	0.52
push,push	2.70%	0.11%	2.49%	1.72%	1.51%	1.98%	1.51%	0.96
lea,mov	2.50%	1.93%	1.52%	1.24%	1.26%	1.35%	1.26%	0.36
test,jne	1.70%	1.19%	1.32%	1.52%	1.60%	1.09%	1.59%	0.32
push,mov	1.60%	1.40%	0.32%	1.30%	1.00%	1.98%	1.00%	0.84
mov,lea	1.40%	1.22%	1.30%	0.92%	0.96%	0.99%	0.96%	0.29
cmp,je	1.10%	0.60%	0.93%	1.14%	1.23%	0.83%	1.23%	0.51
xor,mov	1.10%	0.20%	0.21%	1.43%	1.31%	1.58%	1.31%	0.88
mov,jmpq	1.00%	0.70%	0.72%	1.13%	1.24%	0.57%	1.24%	0.54
mov,add	1.00%	1.58%	0.72%	0.87%	0.82%	0.57%	0.82%	0.64
sub,mov	0.80%	1.57%	0.99%	0.85%	0.76%	0.49%	0.76%	0.69
add,mov	0.70%	1.66%	0.82%	0.71%	0.68%	0.51%	0.68%	0.70
jmpq,mov	0.70%	0.90%	0.96%	0.78%	0.87%	0.66%	0.87%	0.31
mov,cmp	0.70%	1.14%	0.94%	0.62%	0.71%	0.69%	0.71%	0.46
mov,sub	0.60%	1.52%	0.39%	0.71%	0.65%	0.57%	0.65%	0.74
cmp,jne	0.50%	0.69%	1.09%	0.64%	0.67%	0.81%	0.67%	0.42
mov,push	0.50%	0.62%	0.08%	0.88%	0.71%	1.31%	0.71%	0.94
jmp,mov	0.50%	1.52%	1.50%	0.13%	0.13%	0.97%	0.13%	0.91
xor,callq	0.40%	0.01%	0.02%	1.01%	0.99%	1.09%	0.99%	0.99
mov,jmp	0.40%	1.08%	1.04%	0.42%	0.38%	0.76%	0.38%	0.64
pop,retq	0.30%	0.33%	0.70%	0.76%	0.60%	0.84%	0.60%	0.61
OTHER	6.10%	32.18%	40.04%	45.44%	45.84%	44.86%	45.83%	0.30

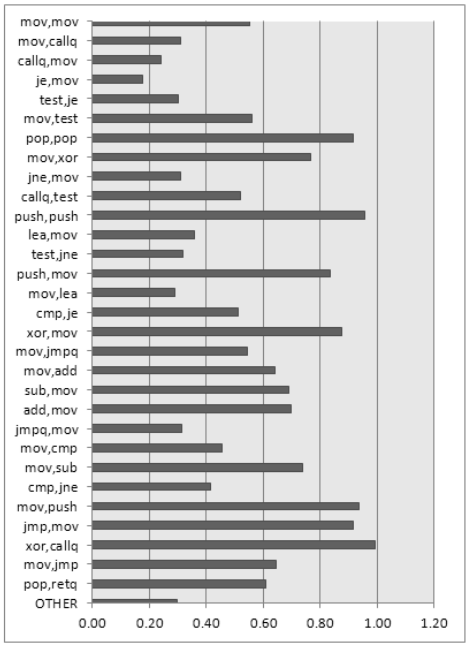
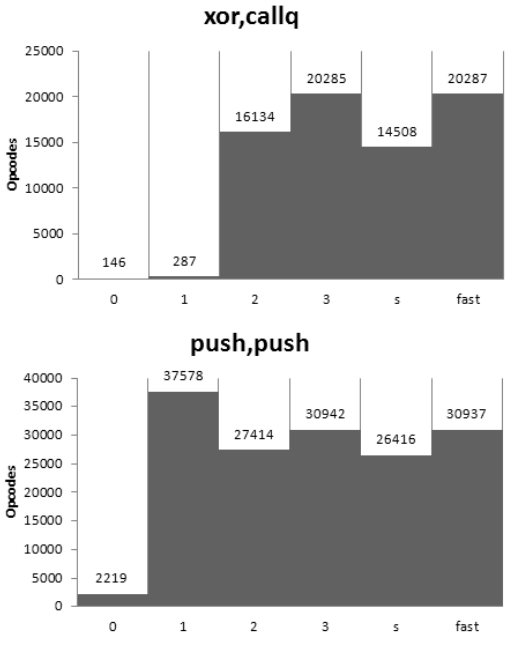


Figure 9: Z-scores and the 2 greatest deviators for different Flags (2-gram).

Opcode	0	1	2	3	s	fast
mov,mov	1.95	-0.14	-0.6	-0.17	-0.87	-0.17
mov,callq	1.67	0.33	-0.91	0.02	-1.13	0.02
callq,mov	1.37	0.21	-0.87	0.37	-1.44	0.37
je,mov	0.9	-0.75	-0.56	0.87	-1.34	0.87
test,je	-0.34	-0.39	-0.33	1.19	-1.33	1.19
mov,test	-0.23	-0.39	-0.06	1.11	-1.53	1.11
pop,pop	-1.9	-0.19	0.44	0.76	0.13	0.76
mov,xor	-1.15	-1.35	0.44	0.9	0.26	0.9
jne,mov	1.73	-0.44	-0.78	0.26	-1.03	0.26
callq,test	-1.04	-1.24	-0.11	1.07	0.26	1.07
push,push	-1.93	0.95	0.12	0.41	0.04	0.41
lea,mov	1.88	-0.31	-0.72	0.04	-0.93	0.04
test,jne	0.02	-0.69	-0.07	1.1	-1.45	1.09
push,mov	1.06	-1.83	0.04	0.01	0.72	0.01
mov,lea	1.55	0.21	-0.92	0.21	-1.27	0.22
cmp,je	-0.83	-0.58	0.07	1.19	-1.05	1.2
xor,mov	-1.22	-1.31	0.49	0.85	0.34	0.85
mov,jmpq	-0.3	-0.83	0.15	1.13	-1.28	1.13
mov,add	1.87	-0.64	-0.29	0.03	-1	0.03
sub,mov	1.88	-0.18	-0.34	-0.1	-1.14	-0.1
add,mov	1.96	-0.32	-0.43	-0.16	-0.9	-0.16
jmpq,mov	0.99	-0.12	-0.67	0.7	-1.6	0.71
mov,cmp	1.81	-0.03	-0.87	0.04	-0.99	0.04
mov,sub	1.93	-0.85	-0.28	-0.06	-0.68	-0.06
cmp,jne	0.51	1.39	-1.29	0.21	-1.02	0.21
mov,push	0.11	-1.97	0.28	0.35	0.87	0.35
jmp,mov	1.55	0.82	-0.83	-0.79	0.04	-0.79
xor,callq	-1.26	-1.24	0.45	0.89	0.27	0.89
mov,jmp	1.74	0.64	-0.83	-0.64	-0.27	-0.64
pop,retq	-1.93	-0.15	0.61	0.65	0.17	0.64
OTHER	-0.47	-0.91	-0.14	1.23	-0.95	1.23



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 build automation tools might be the key to this.

After the dataset has been created, techniques can be applied that proved to be successful for detecting malware. F.g in the research of Hashemi et al [14] the opcodes (2-gram) were transformed into graphs, which were then turned into feature vectors. The feature vectors were then used to classify between malware or benign.

Also, experimentation with different sorts of classifiers can be done. To test the effectiveness of some of the more successful classifiers that were mentioned in the related works section (DT (Random Forest), BDT, PART, KNN, BN, SVM and Adaboost).

Aside from using opcodes, exploring other artefacts of the binary are also possible such as the appearance of combinations of bytes or hexadecimals.

Currently, the measurements have been done on a collection of binaries. But research can also be done on the effects of different GCC flags and versions per application. 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 Software 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

[1] G. Robles, J. M. Gonzalez-Barahona, and I. Herraiz, "An empirical approach to software archaeology," in *Proc. of 21st Int. Conf. on Software Maintenance (ICSM 2005), Budapest, Hungary*, 2005, pp. 47–50.

[2] D. Bilar, "Opcodes as predictor for malware," vol. 1, 01 2007.

[3] D. Bilar *et al.*, "Statistical structures: Fingerprinting malware for classification and analysis," *Proceedings of Black Hat Federal 2006*, 2006.

[4] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, "Idea: Opcode-sequence-based malware detection," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2010, pp. 35–43.

[5] T. H. Austin, E. Filiol, S. Josse, and M. Stamp, "Exploring hidden markov models for virus analysis: a semantic approach," in *System Sciences (HICSS), 2013 46th Hawaii International Conference on*. IEEE, 2013, pp. 5039–5048.

[6] B. Kang, S. Y. Yerima, S. Sezer, and K. McLaughlin, "N-gram opcode analysis for android malware detection," *arXiv preprint arXiv:1612.01445*, 2016.

[7] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas, "Opcode sequences as representation of executables for data-mining-based unknown malware detection," *Information Sciences*, vol. 231, pp. 64–82, 2013.

[8] W. Wong and M. Stamp, "Hunting for metamorphic engines," *Journal in Computer Virology*, vol. 2, no. 3, pp. 211–229, 2006.

[9] A. H. Toderici and M. Stamp, "Chi-squared distance and metamorphic virus detection," *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 1, pp. 1–14, 2013.

[10] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, "Graph-based malware detection using dynamic analysis," *Journal in computer Virology*, vol. 7, no. 4, pp. 247–258, 2011.

[11] N. Runwal, R. M. Low, and M. Stamp, "Opcode graph similarity and metamorphic detection," *Journal in Computer Virology*, vol. 8, no. 1-2, pp. 37–52, 2012.

[12] S. Deshpande, Y. Park, and M. Stamp, "Eigenvalue analysis for metamorphic detection," *Journal of computer virology and hacking techniques*, vol. 10, no. 1, pp. 53–65, 2014.
[13] M. E. Saleh, A. B. Mohamed, and A. A. Nabi, "Eigenviruses for metamorphic virus recognition," *IET information security*, vol. 5, no. 4, pp. 191–198, 2011.
[14] H. Hashemi, A. Azmoodeh, A. Hamzeh, and S. Hashemi, "Graph embedding as a new approach for unknown malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 3, pp. 153–166, 2017.
[15] M. Eskandari, Z. Khorshidpour, and S. Hashemi, "Hdm-analyser: a hybrid analysis approach based on data mining techniques for malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 2, pp. 77–93, 2013.
[16] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, "Detecting unknown malicious code by applying classification techniques on opcode patterns," *Security Informatics*, vol. 1, no. 1, p. 1, 2012.
[17] M. Fazlali, P. Khodamoradi, F. Mardukhi, M. Nosrati, and M. M. Dehshibi, "Metamorphic malware detection using opcode frequency rate and decision tree," *International Journal of Information Security and Privacy (IJISP)*, vol. 10, no. 3, pp. 67–86, 2016.
[18] G. Shanmugam, R. M. Low, and M. Stamp, "Simple substitution distance and metamorphic detection," *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 3, pp. 159–170, 2013.
[19] I. Sorokin, "Comparing files using structural entropy," *Journal in computer virology*, vol. 7, no. 4, pp. 259–265, 2011.
[20] gcc.gnu.org. Using the gnu compiler collection (gcc): Optimize options. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
[21] "Standarding - Z Scores in Excel," Feb 2018, [Online; accessed 10. Feb. 2018]. [Online]. Available: https://www.youtube.com/watch?v=tkkxIPAysME
[22] "Excel - Pearson chi square test of independence," Feb 2018, [Online; accessed 10. Feb. 2018]. [Online]. Available: https://www.youtube.com/watch?v=dgjHsv8FBYU
[23] "Excel - Cramer's V," Feb 2018, [Online; accessed 10. Feb. 2018]. [Online]. Available: https://www.youtube.com/watch?v=YXe51-N9xjM
[24] "Statistical Interpretation | Fort Collins Science Center," Jan 2018, [Online; accessed 24. Jan. 2018]. [Online]. Available: https://www.fort.usgs.gov/sites/landsat-imagery-unique-resource/statistical-interpretation
[25] "VX Heaven Virus Collection 2010-05-18," Jan 2018, [Online; accessed 30. Jan. 2018]. [Online]. Available: http://academictorrents.com/details/34ebe49a48aa532deb9c0dd08a08a017aa04d810/tech&dllist=1

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).

Figure 1: Relative frequencies of opcodes between the individual applications of the dataset.

Pearson's chi-squared test (χ2)		184522.4	GCC: (Ubuntu 5.4.1-2ubuntu1~16.04) 5.4.1 20160904																		
Cramér's V		0.054752																			
p		0																			
Opcode	**Average**	**barcode**	**bash**	**cp**	**enscript**	**find**	**gap**	**gcal**	**gcal2txt**	**git**	**git-shell**	**lighttpd**	**locate**	**ls**	**mv**	**openssl**	**postgres**	**sha256sum**	**sha384sum**	**units**	**vim**
mov	34.00%	27.90%	31.20%	31.80%	34.70%	30.80%	35.30%	40.70%	33.90%	33.60%	33.30%	35.80%	29.90%	29.30%	31.70%	32.80%	36.10%	32.80%	33.30%	28.60%	30.00%
callq	8.30%	7.30%	8.50%	7.60%	8.80%	7.20%	6.20%	8.00%	10.00%	9.20%	8.40%	7.90%	6.90%	5.90%	7.60%	7.50%	9.50%	4.20%	3.70%	9.70%	7.90%
test	4.90%	4.80%	6.20%	4.60%	6.20%	4.50%	4.70%	2.10%	2.30%	5.40%	5.30%	4.80%	4.40%	4.40%	4.60%	5.00%	4.30%	2.00%	1.80%	5.90%	6.40%
je	4.70%	4.50%	6.40%	4.90%	5.60%	4.50%	4.20%	2.10%	5.00%	4.50%	4.60%	5.30%	4.40%	4.40%	5.00%	4.80%	4.20%	2.10%	1.90%	5.50%	6.20%
xor	4.60%	3.30%	3.90%	4.80%	4.10%	4.80%	3.20%	4.10%	4.40%	4.80%	4.60%	4.30%	4.60%	4.20%	4.90%	5.60%	5.10%	10.30%	10.60%	3.80%	3.90%
pop	3.40%	3.80%	3.30%	3.90%	1.70%	4.10%	3.40%	2.30%	3.00%	3.50%	4.20%	4.30%	3.10%	3.60%	3.90%	3.20%	3.70%	2.40%	2.10%	4.00%	3.20%
cmp	3.30%	2.20%	3.80%	3.80%	2.30%	3.60%	5.20%	2.20%	5.80%	2.90%	3.10%	3.60%	3.80%	5.20%	4.00%	2.30%	2.80%	2.50%	2.20%	1.90%	4.60%
jne	3.20%	2.60%	3.70%	3.30%	2.80%	3.00%	4.60%	1.60%	2.90%	3.70%	3.50%	2.70%	3.40%	3.10%	3.30%	2.20%	2.70%	2.10%	1.80%	3.20%	3.90%
lea	3.10%	6.20%	2.40%	2.30%	5.60%	2.50%	3.40%	2.20%	2.80%	3.80%	3.50%	2.30%	3.10%	2.50%	2.20%	2.90%	2.90%	3.60%	2.70%	8.00%	3.00%
jmpq	3.00%	3.40%	3.80%	4.60%	4.60%	4.00%	3.00%	1.90%	8.50%	2.60%	2.60%	3.50%	5.10%	4.90%	4.60%	2.80%	2.70%	3.40%	3.00%	4.20%	4.10%
push	3.00%	2.70%	2.40%	2.70%	1.40%	3.00%	2.20%	10.90%	3.80%	2.90%	3.30%	3.00%	2.60%	2.70%	2.70%	2.30%	2.90%	1.90%	1.60%	3.50%	2.20%
add	2.80%	4.30%	2.70%	2.80%	3.20%	2.70%	3.20%	3.30%	0.80%	2.70%	3.00%	3.10%	3.00%	2.90%	2.80%	3.20%	2.40%	6.00%	8.00%	2.40%	2.90%
nopl	2.20%	1.80%	2.70%	2.40%	1.40%	2.80%	2.50%	0.70%	1.70%	2.20%	2.40%	2.20%	2.40%	2.60%	2.50%	2.00%	2.20%	1.60%	1.40%	2.40%	2.40%
sub	1.60%	2.30%	1.40%	1.50%	1.20%	1.70%	1.60%	3.10%	1.00%	1.60%	1.80%	1.50%	1.50%	2.00%	1.50%	1.30%	1.30%	1.00%	0.80%	1.30%	1.70%
nopw	1.40%	1.10%	1.60%	1.50%	0.70%	1.70%	1.50%	0.40%	1.10%	1.40%	1.50%	1.40%	1.50%	1.90%	1.60%	1.40%	1.30%	1.00%	0.90%	1.40%	1.40%
retq	1.10%	1.10%	1.30%	1.40%	0.40%	1.70%	1.00%	0.30%	0.80%	1.10%	1.30%	1.20%	1.10%	1.50%	1.40%	1.20%	1.20%	0.80%	0.70%	1.10%	1.10%
movl	1.00%	1.00%	1.30%	0.60%	0.70%	0.60%	0.10%	1.00%	1.20%	1.40%	0.80%	1.30%	0.50%	0.60%	0.60%	0.90%	1.00%	0.90%	0.20%	1.20%	1.90%
jmp	1.00%	1.10%	1.20%	0.90%	0.70%	1.30%	0.90%	0.30%	0.80%	1.10%	1.20%	0.90%	1.00%	1.30%	0.90%	0.90%	0.90%	0.50%	0.40%	1.00%	1.20%
movq	0.90%	0.40%	0.70%	0.90%	0.30%	1.00%	1.10%	0.30%	0.00%	2.20%	1.30%	0.90%	1.40%	0.60%	0.90%	0.90%	0.80%	0.60%	0.60%	0.50%	0.60%
movzbl	0.90%	1.10%	1.00%	1.10%	0.60%	0.80%	0.50%	0.40%	0.10%	0.70%	0.80%	0.50%	1.50%	1.30%	1.00%	1.10%	0.90%	0.90%	0.80%	0.40%	1.10%
and	0.70%	1.00%	0.80%	1.10%	0.40%	0.80%	1.00%	0.10%	0.20%	0.70%	0.70%	0.50%	1.10%	0.80%	1.10%	1.00%	0.50%	2.90%	3.00%	0.20%	0.50%
movslq	0.60%	0.70%	1.20%	0.10%	1.70%	0.50%	0.10%	0.40%	0.80%	0.50%	0.70%	0.20%	0.30%	0.40%	0.10%	0.50%	0.80%	0.10%	0.10%	0.50%	1.00%
cmpb	0.60%	0.30%	0.90%	1.40%	0.30%	0.90%	0.10%	0.30%	0.30%	0.60%	0.60%	0.30%	1.20%	1.20%	1.30%	0.20%	0.80%	0.90%	0.80%	0.60%	0.70%
jle	0.50%	0.40%	0.50%	0.10%	0.50%	0.10%	0.40%	0.40%	0.30%	0.30%	0.30%	0.30%	0.10%	0.20%	0.10%	0.60%	0.40%	0.00%	0.00%	0.20%	0.80%
movb	0.50%	0.60%	0.70%	1.20%	0.70%	1.40%	0.10%	0.20%	1.00%	0.40%	0.40%	0.20%	2.90%	1.20%	1.00%	0.20%	0.70%	1.10%	1.00%	0.40%	0.40%
shr	0.40%	0.30%	0.10%	0.30%	0.40%	0.30%	0.80%	0.10%	0.10%	0.30%	0.20%	0.10%	0.40%	0.30%	0.30%	0.90%	0.30%	1.40%	1.60%	0.10%	0.20%
movzwl	0.40%	0.30%	0.00%	0.10%	0.00%	0.10%	1.80%	0.00%	0.00%	0.10%	0.10%	0.40%	0.10%	0.10%	0.10%	0.10%	0.50%	0.00%	0.00%	0.00%	0.10%
nop	0.40%	0.30%	0.40%	0.30%	0.20%	0.40%	0.40%	0.10%	0.20%	0.40%	0.40%	0.40%	0.40%	0.30%	0.30%	0.50%	0.30%	0.10%	0.10%	0.30%	0.40%
cmpq	0.40%	0.30%	0.30%	0.40%	0.40%	0.30%	0.80%	0.10%	0.20%	0.30%	0.30%	0.80%	0.10%	0.30%	0.50%	0.30%	0.20%	0.10%	0.10%	0.40%	0.40%
shl	0.30%	0.00%	0.20%	0.30%	0.10%	0.20%	0.70%	0.00%	0.00%	0.30%	0.30%	0.30%	0.30%	0.20%	0.30%	0.30%	0.40%	0.10%	0.10%	0.10%	0.30%
OTHER	6.90%	12.90%	5.30%	7.30%	6.90%	8.50%	5.80%	10.70%	7.30%	5.00%	5.40%	6.10%	7.90%	10.30%	7.30%	11.20%	6.20%	12.70%	14.50%	7.00%	5.60%

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.

Figure 2: Absolute number of opcodes between the individual applications of the dataset.

Opcode	**Total**	barcode	bash	cp	enscript	find	gap	gcal	gcal2txt	git	git-shell	lighttpd	locate	ls	mv	openssl	postgres	sha256sum	sha384sum	units	vim	**μ Mean**	**σ Std. Dev.**
TOTAL	3480062	8784	168498	22156	20586	31184	358026	147410	1331	380193	209955	43218	10814	19240	22281	523024	1009098	8535	9720	12936	473073	174003	259775
mov	1182654	2454	52492	7043	7146	9608	126294	59938	451	127844	69895	15481	3233	5643	7063	171813	364577	2797	3240	3703	141939	59133	90827
callq	288090	637	14253	1681	1820	2233	22115	11754	133	34837	17589	3415	743	1144	1695	39350	95477	357	357	1257	37243	14405	23266
test	171085	419	10449	1026	1279	1399	16755	3035	30	20630	11181	2068	477	842	1026	26031	42935	173	173	768	30389	8554	12445
je	162024	398	10812	1093	1145	1412	15182	3025	66	17146	9576	2282	472	841	1117	24933	42336	183	183	712	29110	8101	11929
xor	161214	287	6542	1055	835	1503	11369	5984	58	18424	9620	1850	497	801	1091	29459	50991	875	1032	495	18446	8061	12897
pop	119301	336	5587	863	352	1272	12006	3335	40	13481	8747	1868	331	693	863	16536	36854	206	206	517	15208	5965	9188
cmp	115549	196	6382	850	464	1133	18790	3175	77	10916	6509	1545	412	991	899	12195	28479	213	210	240	21873	5777	8407
jne	109689	228	6176	736	568	939	16385	2287	39	13970	7344	1180	371	605	738	11296	27403	178	178	419	18649	5484	7914
lea	106653	541	3978	511	1150	775	12195	3239	37	14513	7432	1007	330	486	496	15042	29278	306	264	1038	14035	5333	7807
jmpq	105581	295	6416	1010	951	1241	10917	2830	113	9872	5484	1495	555	937	1026	14682	27366	293	293	542	19263	5279	7548
push	103661	234	4026	602	288	947	7975	16062	51	10923	7022	1287	283	510	598	12072	29366	160	160	458	10637	5183	7589
add	99024	381	4572	617	659	844	11469	4927	10	10122	6237	1335	324	555	615	16715	24306	510	776	310	13740	4951	6852
nopl	76695	162	4623	539	295	886	9005	1071	22	8338	5033	967	262	498	554	10523	21853	138	139	316	11471	3835	5745
sub	54054	202	2339	335	241	541	5811	4610	13	5924	3780	666	165	386	328	7011	13497	83	81	162	7879	2703	3688
nopw	47579	96	2780	341	151	522	5527	611	15	5281	3226	586	165	362	353	7225	13135	82	85	187	6849	2379	3528
retq	38870	95	2250	309	91	523	3614	479	10	4166	2749	522	115	280	313	6025	11874	70	70	139	5176	1944	3015
movl	35944	91	2188	130	134	192	503	1518	16	5218	1722	574	57	109	128	4476	9807	76	16	157	8832	1797	2970
jmp	34158	100	1997	208	135	401	3306	377	11	4334	2618	389	104	250	199	4513	9508	41	41	126	5500	1708	2554
movq	33006	39	1108	200	55	313	4014	453	0	8204	2723	369	150	119	206	4482	7709	50	62	63	2687	1650	2564
movzbl	30981	95	1739	236	130	265	1721	521	1	2734	1686	215	159	254	231	5892	9572	76	77	52	5325	1549	2551
and	23749	85	1334	249	87	245	3512	170	3	2583	1538	207	118	148	234	5341	5008	244	292	25	2326	1187	1692
movslq	22297	64	2045	19	340	162	400	526	11	2018	1399	93	37	86	19	2560	7798	6	6	67	4641	1115	1985
cmpb	20106	25	1496	317	67	279	212	488	4	2258	1199	136	129	234	299	947	8255	75	75	81	3530	1005	1929
jle	16343	31	872	14	104	45	1567	576	4	1238	703	109	10	35	20	2971	4392	4	4	29	3615	817	1327
movb	16113	57	1193	256	140	438	490	281	13	1538	857	70	314	228	222	955	7082	95	95	50	1739	806	1564
shr	14618	28	245	57	81	88	2895	86	1	1043	509	61	47	57	57	4930	3361	121	156	8	787	731	1365
movzwl	13004	23	41	16	1	36	6378	24	0	193	170	170	9	12	19	623	5008	3	3	0	275	650	1745
nop	12875	24	702	73	45	124	1334	149	3	1333	812	165	42	64	70	2698	3453	12	14	34	1724	644	992
cmpq	12539	25	508	93	90	101	3037	91	2	1125	560	341	16	56	107	1830	2387	12	12	57	2089	627	940
shl	12167	2	402	61	27	76	2603	43	0	1130	670	127	32	33	61	1492	3743	11	13	13	1628	608	1031
OTHER	240439	1134	8951	1616	1715	2641	20645	15745	97	18857	11365	2638	855	1981	1634	58406	62288	1085	1407	911	26468	12022	18313

The absolute values of the opcodes for each application. The rightmost two column 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).

Opcode	Total	GCC 4.4	GCC 4.6	GCC 4.7	GCC 4.8	GCC 4.9	GCC 5	GCC 6	GCC 7	μ Mean	σ Std. Dev.
TOTAL	27444775	3392321	3360524	3353191	3413109	3475608	3480062	3478506	3491454	3430597	57501
mov	9584697	1247531	1228735	1214602	1169725	1179637	1182654	1185099	1176714	1198087	28449
callq	2275685	279019	283254	278324	280166	288731	288090	288411	289690	284461	4804
test	1362558	167599	171673	168097	168868	169086	171085	172107	174043	170320	2250
je	1292616	160951	161700	162672	159732	159033	162024	162362	164142	161577	1642
xor	1249100	149826	156482	148695	151830	161130	161214	159554	160369	156138	5267
cmp	910733	113023	111314	112561	112815	114416	115549	114831	116224	113842	1676
jne	843922	99233	100720	100558	102471	109465	109689	109645	112141	105490	5214
jmpq	828846	102477	97840	103844	103719	104996	105581	105265	105124	103606	2549
lea	813129	88951	91193	96360	106133	108272	106653	106351	109216	101641	8167
pop	807013	69445	69674	70419	125242	120315	119301	116201	116416	100877	25847
add	803482	101245	105546	103506	97940	100008	99024	98812	97401	100435	2840
push	659402	52495	52906	53288	86302	104466	103661	103225	103059	82425	25146
nopl	630254	86398	75638	77608	79475	77385	76695	76379	80676	78782	3493
sub	419062	51213	50871	50885	48164	53873	54054	54854	55148	52383	2463
nopw	383949	47854	46339	48307	48147	48304	47579	47471	49948	47994	1017
retq	306300	36258	36272	37414	41542	38965	38870	38366	38613	38288	1709
movl	299461	39660	39715	39414	40335	35474	35944	35949	32970	37433	2691
jmp	267480	34837	30835	33321	33881	33865	34158	33672	32911	33435	1194
movq	262221	32379	32514	32617	32731	32432	33006	33008	33534	32778	388
movzbl	258713	35356	31987	31212	31750	31770	30981	34034	31623	32339	1530
and	182921	22026	21313	22246	22602	23458	23749	23927	23600	22865	954
movslq	178054	22874	21607	22297	22744	22033	22297	22277	21925	22257	415
cmpb	158016	18318	20049	19987	20089	20335	20106	19741	19391	19752	645
jle	129059	16415	15983	15834	16246	16181	16343	16259	15798	16132	233
movb	126753	15706	16060	16117	16351	15956	16113	16030	14420	15844	603
shr	111433	12711	12951	13693	14502	14399	14618	14584	13975	13929	751
nop	109664	15887	13670	13873	14163	12929	12875	12742	13525	13708	1018
movzwl	103344	12687	13250	12710	13026	12857	13004	12885	12925	12918	182
cmpq	100827	12875	12613	13581	12449	12564	12539	12582	11624	12603	537
shl	100454	13623	12802	12615	12462	12980	12167	11836	11969	12557	586
OTHER	1885627	233449	225018	226534	227507	240293	240439	240047	252340	235703	9332

The absolute values of the opcodes for different compiler versions. The rightmost two column 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).

Opcode	Total	GCC 4.4	GCC 4.6	GCC 4.7	GCC 4.8	GCC 4.9	GCC 5	GCC 6	GCC 7	μ Mean	σ Std. Dev.
TOTAL	27444639	3392304	3360507	3353174	3413092	3475591	3480045	3478489	3491437	3430580	57501
mov,mov	4254141	594883	579058	568760	506657	501511	504420	504760	494092	531768	41458
mov,callq	1507749	188639	187041	187749	188178	189389	188644	188525	189584	188469	828
callq,mov	1089722	137868	138293	134663	137028	135490	135497	134873	136010	136215	1363
mov,xor	599399	69571	74914	68223	68793	79403	79367	79392	79736	74925	5265
test,je	578598	70495	71716	72046	70998	70841	73880	73904	74718	72325	1621
je,mov	573559	73346	72236	72974	70612	70355	71378	71430	71228	71695	1070
pop,pop	540956	45862	45682	45838	84776	81296	80672	78356	78474	67620	18181
mov,test	442855	53742	52682	53717	56054	56047	57162	57347	56104	55357	1740
callq,test	429928	53887	54319	55024	52835	53483	53260	53415	53705	53741	679
xor,mov	397703	49734	47746	47642	46480	50818	50977	52020	52286	49713	2183
test,jne	351550	43031	43711	43381	43725	44256	45014	44635	43797	43944	654
jne,mov	336187	40766	41910	40728	41282	42081	42940	43008	43472	42023	1051
lea,mov	317622	34979	36690	39186	41586	41494	41330	41018	41339	39703	2549
mov,jmpq	309865	37093	35081	37507	39136	39854	40345	40300	40549	38733	1975
push,push	307297	21428	20300	21033	31911	52974	53131	53234	53286	38412	16171
xor,callq	291240	34449	39732	35064	34740	36733	36687	36794	37041	36405	1699
cmp,je	290811	34424	36328	37456	36385	35998	36941	36740	36539	36351	894
push,mov	259813	22537	23889	22421	40195	39157	37561	37033	37020	32477	7974
nopl,mov	252938	33927	31339	31930	30109	31019	30953	30647	33014	31617	1280
mov,add	244404	37567	37072	36772	26246	26648	27199	26839	26061	30551	5469
test,mov	237724	29077	32202	28951	29471	29962	28133	29257	30671	29716	1249
mov,lea	228584	24558	23860	25572	30789	31074	30671	30907	31153	28573	3273
jmpq,nopl	223756	29479	27027	27501	27930	27212	27313	27078	30216	27970	1209
jmpq,mov	205286	26105	23865	26038	26097	26046	26668	26702	23765	25661	1171
add,mov	190099	21471	23829	22968	25240	24321	24086	24383	23801	23762	1126
pop,retq	189509	17597	18066	18401	29809	26749	26436	25828	26623	23689	4844
mov,je	186497	21804	24091	22930	23165	23714	22785	23329	24679	23312	875
sub,mov	179750	20932	20541	20393	17819	24902	24864	25132	25167	22469	2879
mov,sub	178252	24575	24029	23120	18252	22494	21737	21893	22152	22282	1918
cmp,jne	162239	18403	18595	19691	20596	20660	20926	20847	22521	20280	1348
OTHER	12086606	1480075	1454663	1465495	1516198	1539610	1539068	1538863	1552634	1510826	38441

The absolute values of the 2-gram opcodes for different compiler versions. The rightmost two column 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).

Opcode	Total	0	1	2	3	s	fast	μ Mean	σ Std. Dev.
TOTAL	10620513	2093283	1510912	1592447	2044109	1335385	2044377	1770086	329385
mov	3918049	1045048	562815	521510	675712	436890	676074	653008	213015
callq	880297	158245	139514	131332	163006	125204	162996	146716	16820
je	519586	74237	74563	79616	111070	68959	111141	86598	19281
test	513724	67338	67058	85469	116481	60922	116456	85621	25267
cmp	365857	45764	55739	56137	80956	46227	81034	60976	16133
jne	355218	55414	54085	54713	71934	47160	71912	59203	10289
xor	328627	15356	12327	69431	83059	65438	83016	54771	32501
push	328131	41141	54488	54395	58331	61449	58327	54689	7154
lea	326650	48627	51299	49480	66053	45033	66158	54442	9262
add	295688	65485	43142	46592	57149	26125	57195	49281	13916
jmpq	287308	33106	34110	52645	69319	28938	69190	47885	18456
pop	269530	11285	39072	54086	58139	48827	58121	44922	17966
sub	178050	39524	29230	27790	33762	13946	33798	29675	8739
jmp	173746	48408	37229	16866	19701	31871	19671	28958	12437
movl	146338	30853	22047	21166	27325	17623	27324	24390	4913
nopl	121116	136	120	35370	42671	144	42675	20186	22128
movzbl	112487	34536	15986	13832	21926	4259	21948	18748	10113
movq	109933	18588	17016	16901	21740	13938	21750	18322	3048
retq	108453	24565	16938	17420	16734	16048	16748	18076	3210
movslq	82703	14935	13148	11545	16267	10523	16285	13784	2441
cmpl	77409	23682	20415	4223	5388	18314	5387	12902	8834
and	72521	15907	8414	9932	15050	8151	15067	12087	3630
nopw	70491	122	114	21772	24188	125	24170	11749	12768
cmpb	60111	1569	11207	10691	13982	8689	13973	10019	4612
pushq	49673	10507	7575	7523	8366	7337	8365	8279	1178
cmpq	49205	18311	5808	5292	7120	5552	7122	8201	5015
movb	48278	5797	7801	7587	10854	5323	10916	8046	2403
jle	47475	6835	6912	7418	10441	5433	10436	7913	2065
shl	38481	13098	4710	4340	6711	2903	6719	6414	3589
or	36091	5125	3964	4511	6172	10139	6180	6015	2205
OTHER	649287	119739	94066	92862	124502	93895	124223	108215	16095

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

Opcode	Total	0	1	2	3	s	fast	μ Mean	σ Std. Dev.
TOTAL	10620429	2093269	1510898	1592433	2044095	1335371	2044363	1770072	329385
mov,mov	1775471	584083	275932	207719	270184	167040	270513	295912	147739
mov,callq	617091	140608	110293	82276	103289	77311	103314	102849	22614
callq,mov	411222	85624	71132	57717	73119	50520	73110	68537	12508
je,mov	238289	48860	32112	34059	48559	26154	48545	39715	10134
test,je	237936	36445	35917	36524	51036	26970	51044	39656	9531
mov,test	191456	29031	27038	31143	45602	13056	45586	31909	12350
pop,pop	182677	4287	27759	36491	40931	32291	40918	30446	13792
mov,xor	182107	12616	9554	37155	44216	34388	44178	30351	15447
jne,mov	160128	39694	23386	20805	28621	18994	28628	26688	7502
callq,test	157274	20771	19713	25641	31793	27554	31802	26212	5222
push,push	155506	2219	37578	27414	30942	26416	30937	25918	12252
lea,mov	152462	40309	22943	19688	25721	18041	25760	25410	7941
test,jne	148826	24919	19917	24275	32617	14553	32545	24804	7074
push,mov	121901	29228	4902	20623	20380	26389	20379	20317	8414
mov,lea	112305	25482	19620	14719	19639	13185	19660	18718	4357
cmp,je	106273	12541	14079	18176	25162	11146	25169	17712	6235
xor,mov	104585	4088	3147	22742	26761	21105	26742	17431	10932
mov,jmpq	101706	14713	10852	18021	25278	7562	25280	16951	7351
mov,add	99154	33176	10873	13932	16785	7597	16791	16526	8897
sub,mov	99078	32898	14935	13508	15598	6541	15598	16513	8730
add,mov	93338	34801	12427	11365	13997	6759	13989	15556	9798
jmpq,mov	90200	18882	14551	12430	17754	8811	17772	15033	3884
mov,cmp	86290	23785	14227	9830	14585	9249	14614	14382	5208
mov,sub	83140	31726	5961	11299	13271	7599	13284	13857	9252
cmp,jne	79373	14428	16531	10165	13718	10809	13722	13229	2369
mov,push	74772	13075	1239	14080	14477	17427	14474	12462	5687
jmp,mov	74703	31753	22628	2074	2661	12936	2651	12451	12457
xor,callq	71647	146	287	16134	20285	14508	20287	11941	9364
mov,jmp	70731	22523	15738	6684	7840	10129	7817	11789	6181
pop,retq	65426	6913	10604	12173	12242	11261	12233	10904	2065
OTHER	4475362	673645	605023	723571	937032	599070	937021	745894	155038