Predicting Fuzzer Performance Based On Program Metrics

By Anna Kovalsky

Introduction

Fuzz testing is one of the most popular dynamic analysis methods used today, and while great strides have been made in creating improved fuzzers and in evaluating fuzzers against each other [3,4,10,13], there have been few to no studies done on predicting which fuzzer performance given a set of program metrics.

Knowing this, I have used the data on fuzzer performance from previous works, namely Google's Fuzzbench, along with program metrics data gathered using SourceMonitor and CPPDepend, to create a stepwise regression formula. Using this formula, I created a simple script to predict which fuzzer would provide the best results given a set of program metrics. This tool, along with all the data gathered, is available on github (https://github.com/AIK13/Fuzz-Predictions).

I: Background

Fuzzing, or fuzz testing, is a dynamic software analysis method that attempts to automatically find software vulnerabilities by generating and sending thousands of invalid, unexpected, or random inputs to the program being tested and recording any crashes or interesting behavior caused by said input [9]. Fuzzing is an old method, with the oldest fuzzer--a simple random input generator--dating back to 1983, and the first use of the term dating back to 1988[2]. In general, fuzzers broadly fall into four categories: blackbox fuzzing, grammar-based fuzzing, white-box fuzzing, and greybox fuzzing.

Broadly, these categories differ in how much information the fuzzer requires in order to begin testing, and in how effective they are in testing programs. Blackbox fuzzers require the least information--only needing the program binary. Grammar-based fuzzers require a binary and an input format or grammar. Whitebox fuzzers require the source code and the binary. Finally, greybox fuzzers vary in how much they require, because they use techniques from all the previous categories. Whitebox, greybox, and grammar-based fuzzers are generally the more effective fuzzer types.

There are new fuzzers developed almost every year, and old fuzzers constantly get updates to improve their efficiency. There has also been a growing effort to compare these fuzzers--that is, to find the best fuzzer overall [3,4,10]. Google's Fuzzbench is a tool that can help compare and benchmark fuzzers, and they frequently publish test results. Even a brief glance over their benchmark results reveals differences in fuzzer performance depending on the type of benchmark program used. This intuitively suggests that some fuzzers are best used only for specific types of programs.

II: Problem

Given that there are new fuzzers every year, and that it appears that fuzzing performance varies depending on type of benchmark program, it is natural to assume that fuzzing performance can be predicted based on the type of benchmark program used. However, there does not seem to be any research done into predicting fuzzing performance. Existing works focus on improving fuzzer performance [10] using program metrics or ranking existing fuzzers [3,4], but no existing work develops a regression model or a predicting tool to specifically predict fuzzer performance for many different fuzzers. Shudrak's and Zolotarev's work on improving fuzzing performance [10] seems like it could be indirectly applied to predict fuzzing performance because they compare many metrics and find the one most correlated with fuzzer effectiveness. However, no regression model is presented in their work, and only one fuzzer is tested with those metrics, so it's possible a different metric would be most correlated with performance for other fuzzers.

Previous works that have simply ranked metrics [3,4] also do not present any regression models to predict performance. Fuzzbench is a useful database for performance data of many different fuzzers, which can be used when creating a regression model, however.

Benchmark
bloaty
curl
freetype2
irssi_server
jsoncpp
libjpeg-turbo
libpcap
libpng
libxml2
mbedtls
openssl
openthread
php-src
OSGeo proj4
re2
sqlite3
systemd
vorbis
woff2
wpantund
zlib

This paper aims to compile enough data to create a regression formula for many different fuzzers and then apply those formulas to predict the best fuzzer for each type of program.

III: Contributions

The contributions from this project are as follows:

- This project compiled over 50 different program metrics using SourceMonitor and CPPDepend on 21 programs. The compiled metrics can be used in future works.
- The project introduces several ready-to-use regression formulas created in RStudio using the metrics data and the performance data of 14 different fuzzers from Fuzzbench.
- Finally, this project introduces a Python tool that uses the resulting regression formula for each of the 14 fuzzers to predict fuzzer performance given raw metrics data for any program in CSV format.

IV: Implementation

The finished scripts and raw data can be found here in the Github repository for this project. (https://github.com/AIK13/Fuzz-Predictions).

In this section, I will first discuss how the program metrics were gathered, including how to use SourceMonitor and CPPDepend. Then, I will briefly discuss how to find a stepwise regression function using RStudio, and finally I will discuss the Python tool. The benchmark programs used are shown in the table to the right. These are the same benchmark programs used by Fuzzbench. Finally, detailed information about what each program metric means can be found in Appendix 1.

.

IV,i: SourceMonitor

SourceMonitor is a free tool that scans source code for basic metrics like number of lines of code, statements, functions, calls, methods per class, percent of code that is branches, block depth, and complexity [11]. SourceMonitor also gives various configuration options when running an analysis. For this project, the analysis was run on source code only (i.e. test files were ignored), a modified complexity metric was used, blank lines were ignored for counting lines of code, and continuous header and footer comments were also ignored for lines of code.

The output for each benchmark was extracted into a CSV file to be used for the regression formula later.

IV,ii: CPPDepend

CPPDepend is a static code analysis tool that reads the binary files instead, after they've been compiled by CPPDepend. This means that in order for CPPDepend to work, it needs to know how to compile the benchmarks. The benchmarks used were meant to be compiled on a Linux machine, so some were encountered at this step that will be discussed later.

CPPDepend offers several ways to compile the program, from just using a Visual Studio project file, to using clang to create a build json file, or manually inputting a build path. I decided to create a Visual Studio project file for each benchmark and build using that.

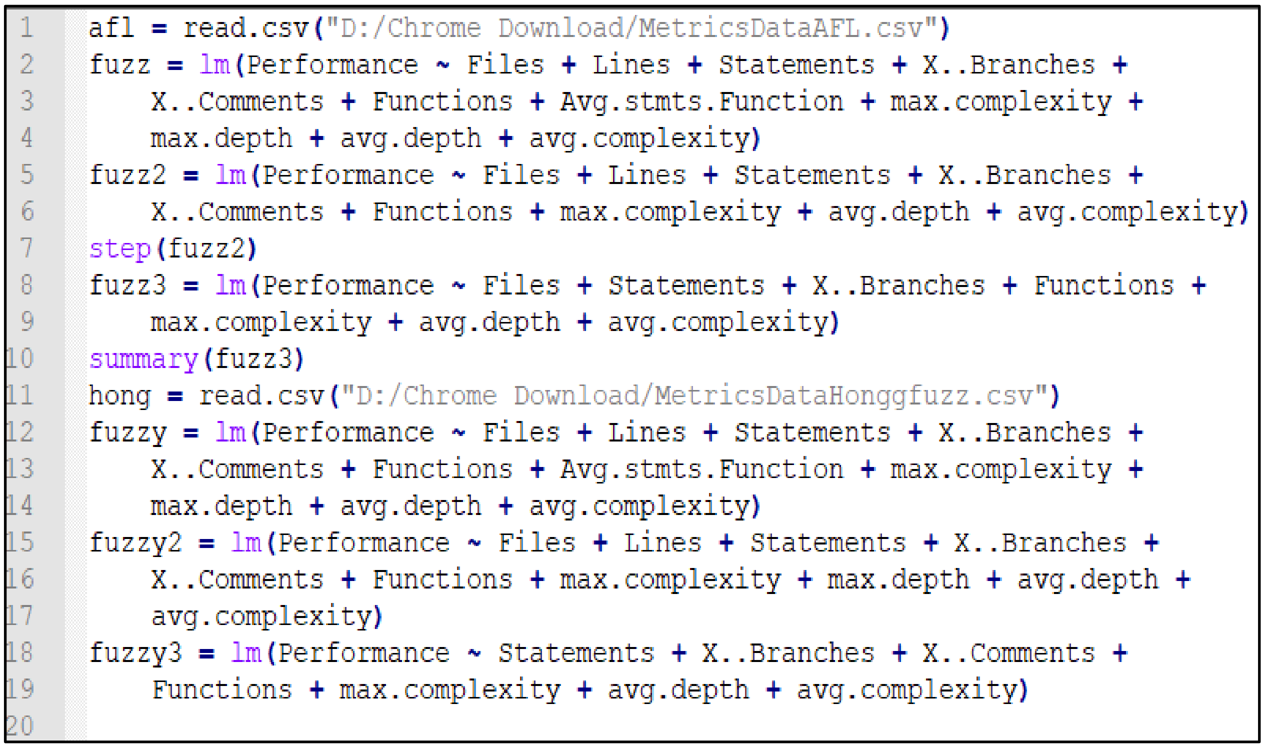
After that, CPPDepend builds and analyzes the binary, and outputs its analysis onto a dashboard. It's possible to export the trend metrics as a CVE. A sample output is shown below, along with the sample CVE output. The CVE output was also stored to be used later for the regression formula.

With CPPDepend, I also gathered another set of metrics. CPPDepend outputs all the rules violations it detects in the program. Rules violations are any violation using CPPDepend standards, or MISRA standard. MISRA are a set of C and C++ coding standards specifically made for embedded tools, but they can also be used for general C and C++ programs. CPPDepend checked MISRA standards automatically, so I included their rules violations at this stage, too. The rules violation metrics could not be easily exported to a CVE, so their summaries were copied into a CVE. Below is a sample table showing program metrics gathered from CPPDepend.

Benchmark	# Lines of Code	# Source Files	# Namespaces	# Types	… 13 more metrics
Bloaty	2897	22	11	105	...

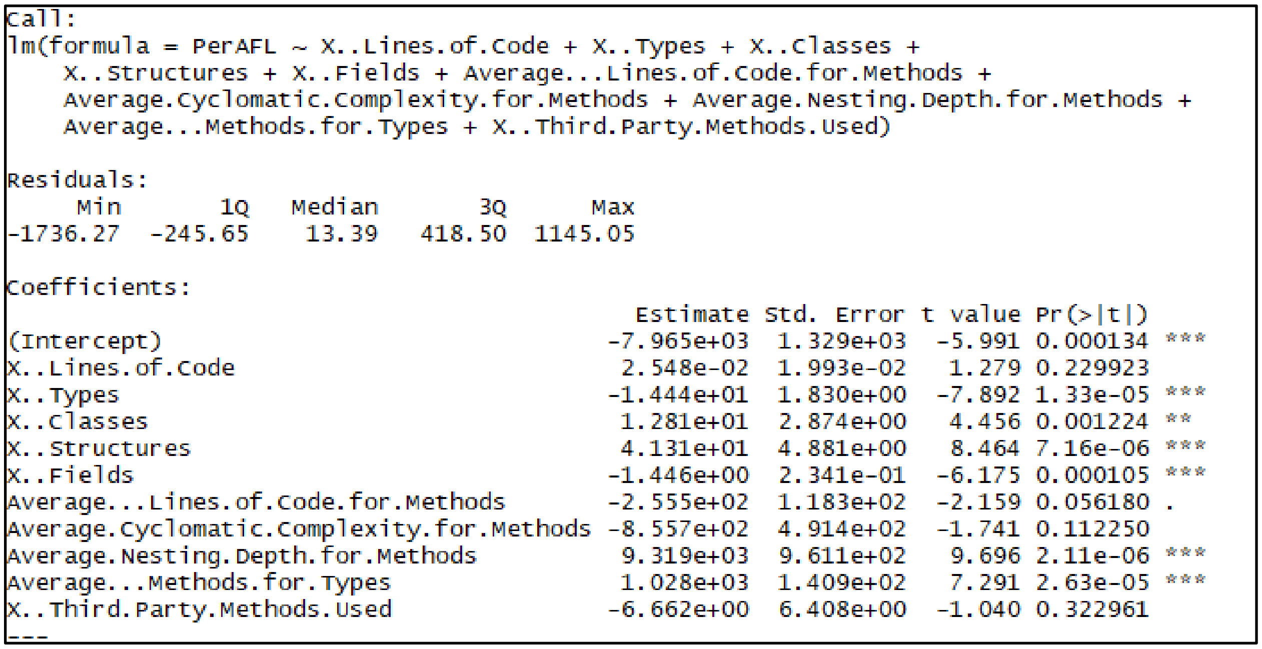
IV,iii: Regression Functions

A stepwise regression is a method to create a regression function where each step deletes an insignificant or irrelevant variable until no more irrelevant variables are left. A regression function is a function that aims to recreate a correlation model. So, in order to do this, the CVEs containing program metrics from SourceMonitor and CPPDepend were input into RStudio after adding columns for fuzzer performance to each CVE. Then, the CVEs were analyzed for any obviously collinear variables, like max complexity and average complexity. Any collinear variables are deleted until only non-collinear variables are left. Then, a regression was run comparing all the metrics against each fuzzer performance. The R code used for this part is shown below.



```r
afl = read.csv("D:/Chrome Download/MetricsDataAFL.csv")
fuzz = lm(Performance ~ Files + Lines + Statements + X..Branches +
    X..Comments + Functions + Avg.stmts.Function + max.complexity +
    max.depth + avg.depth + avg.complexity)
fuzz2 = lm(Performance ~ Files + Lines + Statements + X..Branches +
    X..Comments + Functions + max.complexity + avg.depth + avg.complexity)
step(fuzz2)
fuzz3 = lm(Performance ~ Files + Statements + X..Branches + Functions +
    max.complexity + avg.depth + avg.complexity)
summary(fuzz3)
hong = read.csv("D:/Chrome Download/MetricsDataHonggfuzz.csv")
fuzzy = lm(Performance ~ Files + Lines + Statements + X..Branches +
    X..Comments + Functions + Avg.stmts.Function + max.complexity +
    max.depth + avg.depth + avg.complexity)
fuzzy2 = lm(Performance ~ Files + Lines + Statements + X..Branches +
    X..Comments + Functions + max.complexity + max.depth + avg.depth +
    avg.complexity)
fuzzy3 = lm(Performance ~ Statements + X..Branches + X..Comments +
    Functions + max.complexity + avg.depth + avg.complexity)
```

After each run, RStudio returns the estimates, standard error, t value, and p value for each relevant variable shown below, along with a regression formula.



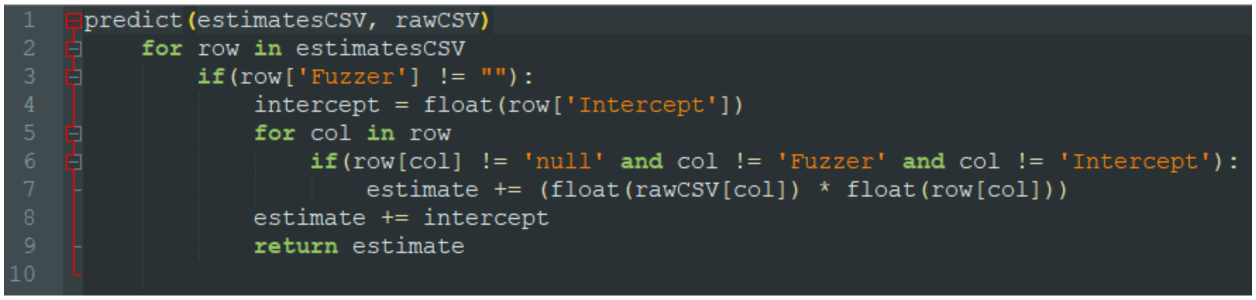
```
Call:
lm(formula = PerAFL ~ X..Lines.of.Code + X..Types + X..Classes +
    X..Structures + X..Fields + Average...Lines.of.Code.for.Methods +
    Average.Cyclomatic.Complexity.for.Methods + Average.Nesting.Depth.for.Methods +
    Average...Methods.for.Types + X..Third.Party.Methods.Used)

Residuals:
     Min       1Q   Median       3Q      Max
-1736.27  -245.65    13.39   418.50  1145.05

Coefficients:
                                            Estimate Std. Error t value Pr(>|t|)
(Intercept)                               -7.965e+03  1.329e+03  -5.991 0.000134 ***
X..Lines.of.Code                           2.548e-02  1.993e-02   1.279 0.229923
X..Types                                  -1.444e+01  1.830e+00  -7.892 1.33e-05 ***
X..Classes                                 1.281e+01  2.874e+00   4.456 0.001224 **
X..Structures                              4.131e+01  4.881e+00   8.464 7.16e-06 ***
X..Fields                                 -1.446e+00  2.341e-01  -6.175 0.000105 ***
Average...Lines.of.Code.for.Methods       -2.555e+02  1.183e+02  -2.159 0.056180 .
Average.Cyclomatic.Complexity.for.Methods -8.557e+02  4.914e+02  -1.741 0.112250
Average.Nesting.Depth.for.Methods          9.319e+03  9.611e+02   9.696 2.11e-06 ***
Average...Methods.for.Types                1.028e+03  1.409e+02   7.291 2.63e-05 ***
X..Third.Party.Methods.Used               -6.662e+00  6.408e+00  -1.040 0.322961
---
```

How the formula works is each estimate for a variable needs to be multiplied with the raw data for that variable. For example, the estimated value for the number of types in the sample is -14.44. Let's say the raw value for the number of types is 10. That means that in the formula, we would need to multiply -14.44 and 10. We would repeat this multiplication for every variable, and then sum all the products together. Finally, we would add the intercept estimate to the result. This would give us the predicted performance for the fuzzer.

The estimate and intercept values are compiled in a CSV file named SourceMonitorRanked, CPPDependRankedMetrics and CPPDependRankedRules in the Github repository.

IV,iv: Python Script

I created two Python scripts, one for SourceMonitor data and the other for CPPDepend data. Both scripts largely use the same algorithm. The only difference is what type of variables they're configured to read. The general algorithm is shown below:



```
predict(estimatesCSV, rawCSV)
    for row in estimatesCSV
        if(row['Fuzzer'] != ""):
            intercept = float(row['Intercept'])
            for col in row
                if(row[col] != 'null' and col != 'Fuzzer' and col != 'Intercept'):
                    estimate += (float(rawCSV[col]) * float(row[col]))
            estimate += intercept
            return estimate
```

The inputs for predict() are two Python dictionaries. The dictionaries are created from two CSV files. The estimatesCSV file contains all the estimate and intercept values for every fuzzer. The estimates and intercept values are gotten from creating a stepwise regression function. The rawCSV file contains all the raw data gotten from SourceMonitor or CPPDepend. It does not matter if there are more variables or data in the raw file, as long as all the variables found in the estimatesCSV file can also be found in the rawCSV file.

In the algorithm, I used the column names from the estimatesCSV as keys when looking up values from the rawCSV. The script itself is a very simple calculator, but it depends on the column names in both CSVs to match. To make sure they match, the Github repository for this project has sample CSV files for both raw data and estimates data. The estimates data should not be changed by the user unless they rerun a regression and get a better regression formula.

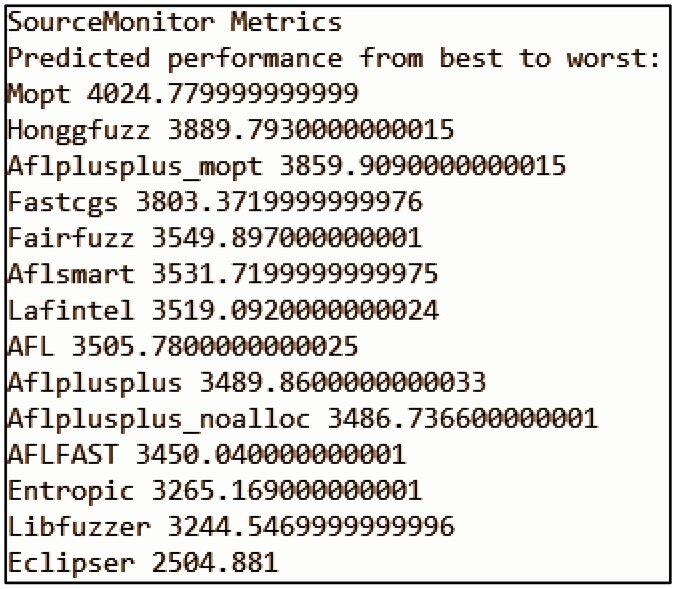
V: Results

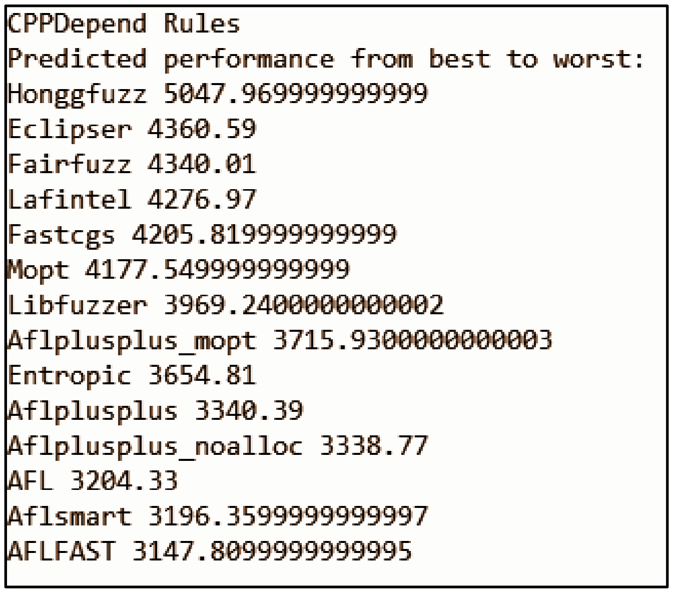
The raw metrics data gathered can be found here (https://github.com/AIK13/Fuzz-Predictions/tree/main/Data). In this section, I will compare the predicted values given by the Python tool and the real values given by Fuzzbench.

V,i: Estimation Evaluation

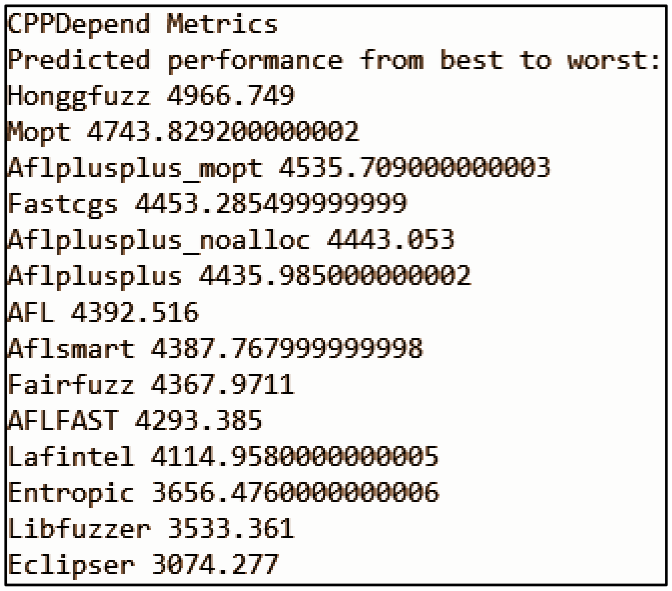
In general, both scripts, when using program metrics, place the top 5 best fuzzers for each benchmark correctly in the top 5. The estimated performance is off by up to 1000, but the order in which the fuzzers are ranked is mostly correct.

As an example, this is the ranked output for bloaty using CPPDepend metrics, SourceMonitor metrics, and CPPDepend rules violations, respectively. On the right are the real performance values for bloaty.

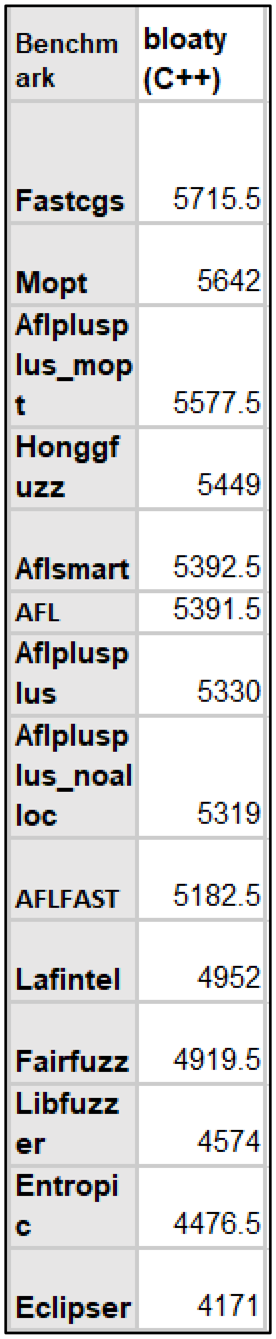

```
SourceMonitor Metrics
Predicted performance from best to worst:
Mopt 4024.779999999999
Honggfuzz 3889.7930000000015
Aflplusplus_mopt 3859.9090000000015
Fastcgs 3803.3719999999976
Fairfuzz 3549.897000000001
Aflsmart 3531.7199999999975
Lafintel 3519.0920000000024
AFL 3505.7800000000025
Aflplusplus 3489.8600000000033
Aflplusplus_noalloc 3486.736600000001
AFLFAST 3450.040000000001
Entropic 3265.169000000001
Libfuzzer 3244.5469999999996
Eclipser 2504.881
```



```
CPPDepend Rules
Predicted performance from best to worst:
Honggfuzz 5047.969999999999
Eclipser 4360.59
Fairfuzz 4340.01
Lafintel 4276.97
Fastcgs 4205.819999999999
Mopt 4177.549999999999
Libfuzzer 3969.2400000000002
Aflplusplus_mopt 3715.9300000000003
Entropic 3654.81
Aflplusplus 3340.39
Aflplusplus_noalloc 3338.77
AFL 3204.33
Aflsmart 3196.3599999999997
AFLFAST 3147.8099999999995
```



```
CPPDepend Metrics
Predicted performance from best to worst:
Honggfuzz 4966.749
Mopt 4743.829200000002
Aflplusplus_mopt 4535.709000000003
Fastcgs 4453.285499999999
Aflplusplus_noalloc 4443.053
Aflplusplus 4435.985000000002
AFL 4392.516
Aflsmart 4387.767999999998
Fairfuzz 4367.9711
AFLFAST 4293.385
Lafintel 4114.9580000000005
Entropic 3656.4760000000006
Libfuzzer 3533.361
Eclipser 3074.277
```



Benchmark	bloaty (C++)
Fastcgs	5715.5
Mopt	5642
Aflplusplus_mopt	5577.5
Honggfuzz	5449
Aflsmart	5392.5
AFL	5391.5
Aflplusplus	5330
Aflplusplus_noalloc	5319
AFLFAST	5182.5
Lafintel	4952
Fairfuzz	4919.5
Libfuzzer	4574
Entropic	4476.5
Eclipser	4171

As you can see from the performance ranking on the right, the correct ranked order is FastCGS, Mopt, AFL++_mopt, Honggfuzz, and AFLSmart. The predicted values are mostly correct for the top 5 when using program metrics. Neither CPPDepend nor SourceMonitor regression functions predict the correct best fuzzer, but both have FastCGS somewhere in the top 5. Meanwhile, using CPPDepend rules mostly fails to predict the best fuzzers.

I checked if this pattern held for the other benchmarks and summarized the results in the tables below.

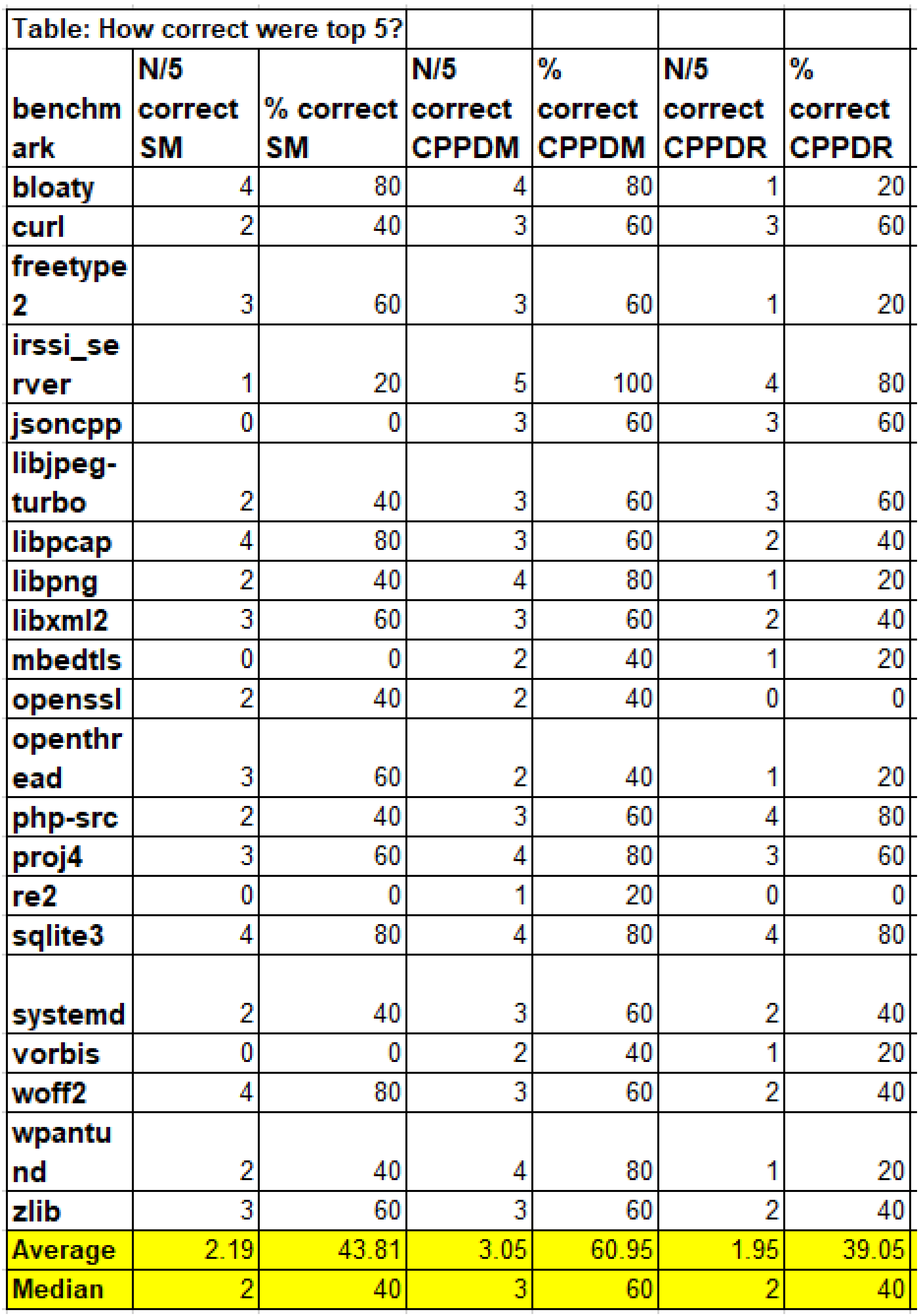


Table: How correct were top 5?						
benchmark	N/5 correct SM	% correct SM	N/5 correct CPPDM	% correct CPPDM	N/5 correct CPPDR	% correct CPPDR
bloaty	4	80	4	80	1	20
curl	2	40	3	60	3	60
freetype2	3	60	3	60	1	20
irssi_server	1	20	5	100	4	80
jsoncpp	0	0	3	60	3	60
libjpeg-turbo	2	40	3	60	3	60
libpcap	4	80	3	60	2	40
libpng	2	40	4	80	1	20
libxml2	3	60	3	60	2	40
mbedtls	0	0	2	40	1	20
openssl	2	40	2	40	0	0
openthread	3	60	2	40	1	20
php-src	2	40	3	60	4	80
proj4	3	60	4	80	3	60
re2	0	0	1	20	0	0
sqlite3	4	80	4	80	4	80
systemd	2	40	3	60	2	40
vorbis	0	0	2	40	1	20
woff2	4	80	3	60	2	40
wpantund	2	40	4	80	1	20
zlib	3	60	3	60	2	40
Average	2.19	43.81	3.05	60.95	1.95	39.05
Median	2	40	3	60	2	40

In the table above, I show how many of the top 5 best performing fuzzers were correctly predicted to be in the top 5 when using Sourcemonitor (SM), CPPDepend metrics (CPPDM), and CPPDepend rules (CPPDR). I also calculated the percentage of the top 5 that were correct. Given this data, it is clear that the regression model using CPPDepend metrics is the most correct, because, on average, it correctly predicted 60.95% of the top 5 best performing fuzzers. Its median for the correct number of fuzzers in the top 5 was also the highest of the three regression models.

Using CPPDepend metrics also provided the closest predicted values for performance in general. SourceMonitor and CPPDepend using rules generally overestimated performance by about 200% on average, while CPPDepend using metrics still overestimated performance on average, but the overestimations were less than 150% generally as you can see in the tables below. However, the predictions were very variable, which implies that there could be some improvements made to the data.

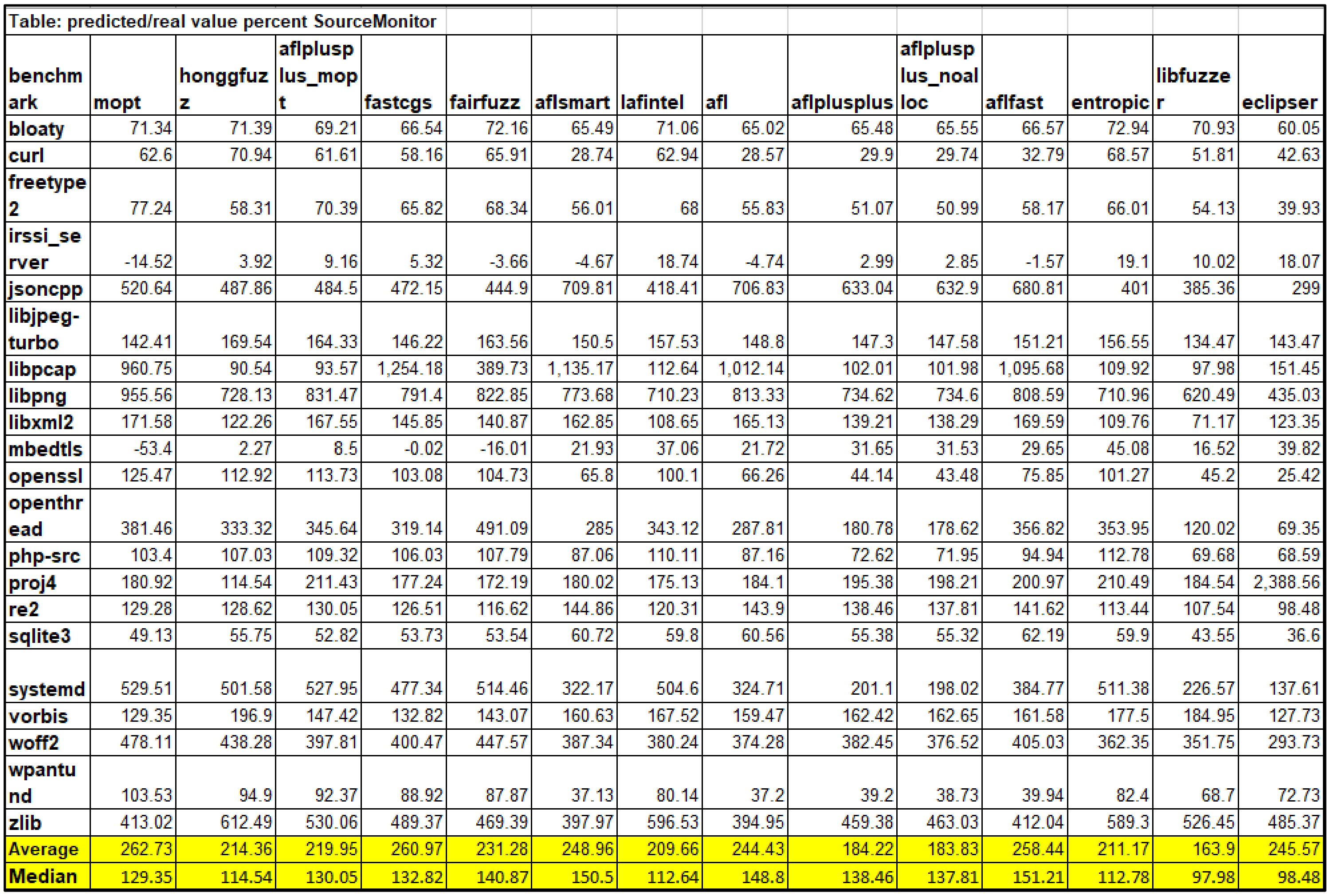


Table: predicted/real value percent SourceMonitor

benchmark	mopt	honggfuzz	aflplusplus_mopt	fastcgs	fairfuzz	aflsmart	lafintel	afl	aflplusplus	aflplusplus_noalloc	aflfast	entropic	libfuzzer	eclipser
bloaty	71.34	71.39	69.21	66.54	72.16	65.49	71.06	65.02	65.48	65.55	66.57	72.94	70.93	60.05
curl	62.6	70.94	61.61	58.16	65.91	28.74	62.94	28.57	29.9	29.74	32.79	68.57	51.81	42.63
freetype2	77.24	58.31	70.39	65.82	68.34	56.01	68	55.83	51.07	50.99	58.17	66.01	54.13	39.93
irssi_server	-14.52	3.92	9.16	5.32	-3.66	-4.67	18.74	-4.74	2.99	2.85	-1.57	19.1	10.02	18.07
jsoncpp	520.64	487.86	484.5	472.15	444.9	709.81	418.41	706.83	633.04	632.9	680.81	401	385.36	299
libjpeg-turbo	142.41	169.54	164.33	146.22	163.56	150.5	157.53	148.8	147.3	147.58	151.21	156.55	134.47	143.47
libpcap	960.75	90.54	93.57	1,254.18	389.73	1,135.17	112.64	1,012.14	102.01	101.98	1,095.68	109.92	97.98	151.45
libpng	955.56	728.13	831.47	791.4	822.85	773.68	710.23	813.33	734.62	734.6	808.59	710.96	620.49	435.03
libxml2	171.58	122.26	167.55	145.85	140.87	162.85	108.65	165.13	139.21	138.29	169.59	109.76	71.17	123.35
mbedtls	-53.4	2.27	8.5	-0.02	-16.01	21.93	37.06	21.72	31.65	31.53	29.65	45.08	16.52	39.82
openssl	125.47	112.92	113.73	103.08	104.73	65.8	100.1	66.26	44.14	43.48	75.85	101.27	45.2	25.42
openthread	381.46	333.32	345.64	319.14	491.09	285	343.12	287.81	180.78	178.62	356.82	353.95	120.02	69.35
php-src	103.4	107.03	109.32	106.03	107.79	87.06	110.11	87.16	72.62	71.95	94.94	112.78	69.68	68.59
proj4	180.92	114.54	211.43	177.24	172.19	180.02	175.13	184.1	195.38	198.21	200.97	210.49	184.54	2,388.56
re2	129.28	128.62	130.05	126.51	116.62	144.86	120.31	143.9	138.46	137.81	141.62	113.44	107.54	98.48
sqlite3	49.13	55.75	52.82	53.73	53.54	60.72	59.8	60.56	55.38	55.32	62.19	59.9	43.55	36.6
systemd	529.51	501.58	527.95	477.34	514.46	322.17	504.6	324.71	201.1	198.02	384.77	511.38	226.57	137.61
vorbis	129.35	196.9	147.42	132.82	143.07	160.63	167.52	159.47	162.42	162.65	161.58	177.5	184.95	127.73
woff2	478.11	438.28	397.81	400.47	447.57	387.34	380.24	374.28	382.45	376.52	405.03	362.35	351.75	293.73
wpantund	103.53	94.9	92.37	88.92	87.87	37.13	80.14	37.2	39.2	38.73	39.94	82.4	68.7	72.73
zlib	413.02	612.49	530.06	489.37	469.39	397.97	596.53	394.95	459.38	463.03	412.04	589.3	526.45	485.37
Average	262.73	214.36	219.95	260.97	231.28	248.96	209.66	244.43	184.22	183.83	258.44	211.17	163.9	245.57
Median	129.35	114.54	130.05	132.82	140.87	150.5	112.64	148.8	138.46	137.81	151.21	112.78	97.98	98.48

Table: predicted/real value percent CPPDepend Metrics

benchmark	mopt	honggfuzz	aflplusplus_mopt	fastcgs	fairfuzz	aflsmart	lafintel	afl	aflplusplus	aflplusplus_noalloc	aflfast	entropic	libfuzzer	eclipser
bloaty	84.08	91.15	81.32	77.92	88.79	81.37	83.1	81.47	83.23	83.53	82.84	81.68	77.25	73.71
curl	104.82	116.98	106.25	105.31	100.96	102.37	106.59	102.01	105.09	104.89	101.42	109.85	109.84	107.95
freetype2	78.32	68.4	85.35	84.41	73.87	87.51	79.99	87.42	86.15	86.33	86.87	81	84.85	82.87
irssi_server	101.39	123.04	95.1	103.58	116.35	111.84	112.4	111.57	105.74	105.49	109.52	111.84	110.87	77.43
jsoncpp	31.45	1.48	42.89	29.97	34.9	62.82	27.94	64.81	102.37	102.34	59.78	56.32	68.91	6.62
libjpeg-turbo	67.87	38.81	45	31.41	28.2	78.15	70.41	77.69	48.04	47.23	72.94	66.12	60.27	-10.35
libpcap	1,628.65	148	118.96	1,130.59	500.78	1,623.74	137.38	1,458.08	163.66	163.42	1,473.70	120.07	123.19	146.22
libpng	-3.96	137.68	85.17	54.76	56.59	13.65	112.73	14.8	92.93	92.85	20.33	126.12	118.88	116.49
libxml2	102.68	82.91	100.75	93.32	86.6	87.55	85.64	88.84	91.4	90.87	89.54	85.65	81.41	97.21
mbedtls	114.42	144.71	135.47	119.25	106.26	129.58	155.53	128.35	126.61	126.77	131.54	155.11	166.42	142.87
openssl	104.22	90.11	85.17	106	97.5	101.7	98.37	101.34	93.1	92.37	101.94	95.74	92.72	85.3
openthread	128.08	94.48	89.36	108.41	127.01	106.03	96.19	105.53	95.9	94.29	108.51	95.12	87.94	83.46
php-src	100.05	102.89	92.92	95.86	97.51	104.93	104.35	104.7	98.69	98.07	105.48	103.72	103.02	75.56
proj4	97.93	105.42	98.42	102.28	99.58	107.92	107.59	107.92	100.71	100.21	109.31	108.11	109.32	-50.66
re2	121.12	102.41	110.28	113.84	110.86	113.04	111.44	113.31	102.83	102.44	111.01	105.05	104.03	92.55
sqlite3	98.4	103.4	96.96	100.83	101.26	102.4	102.59	102.46	101.18	100.99	103.03	102.86	103.66	93.02
systemd	45.3	162.54	49.23	95.09	82.17	171.63	137.78	169.37	119.88	116.29	178.17	136.69	161.74	43.07
vorbis	174.04	159.31	140.33	209.83	217.71	164.2	141.75	162.02	166.03	166.11	169.7	139.59	149.15	140.58
woff2	175.71	197.9	191.03	188.89	197.76	194.15	167.7	190.34	148.98	146.38	206.68	180.02	181.76	205.12
wpantund	108.18	103.79	101.47	113.26	100.45	106.02	111.98	106.25	105.79	105.89	104.16	105.93	107.13	117.49
zlib	-267.73	126.53	-40.76	-205.54	-83.27	-241.4	-49.39	-243.28	-60.06	-60.97	-220.79	-16.37	-6.26	116.53
Average	152.14	109.62	90.98	136.16	111.52	162.34	100.1	154.05	98.96	98.37	157.41	102.39	104.58	87.76
Median	101.39	103.79	95.1	102.28	99.58	104.93	106.59	104.7	101.18	100.99	104.16	105.05	104.03	92.55

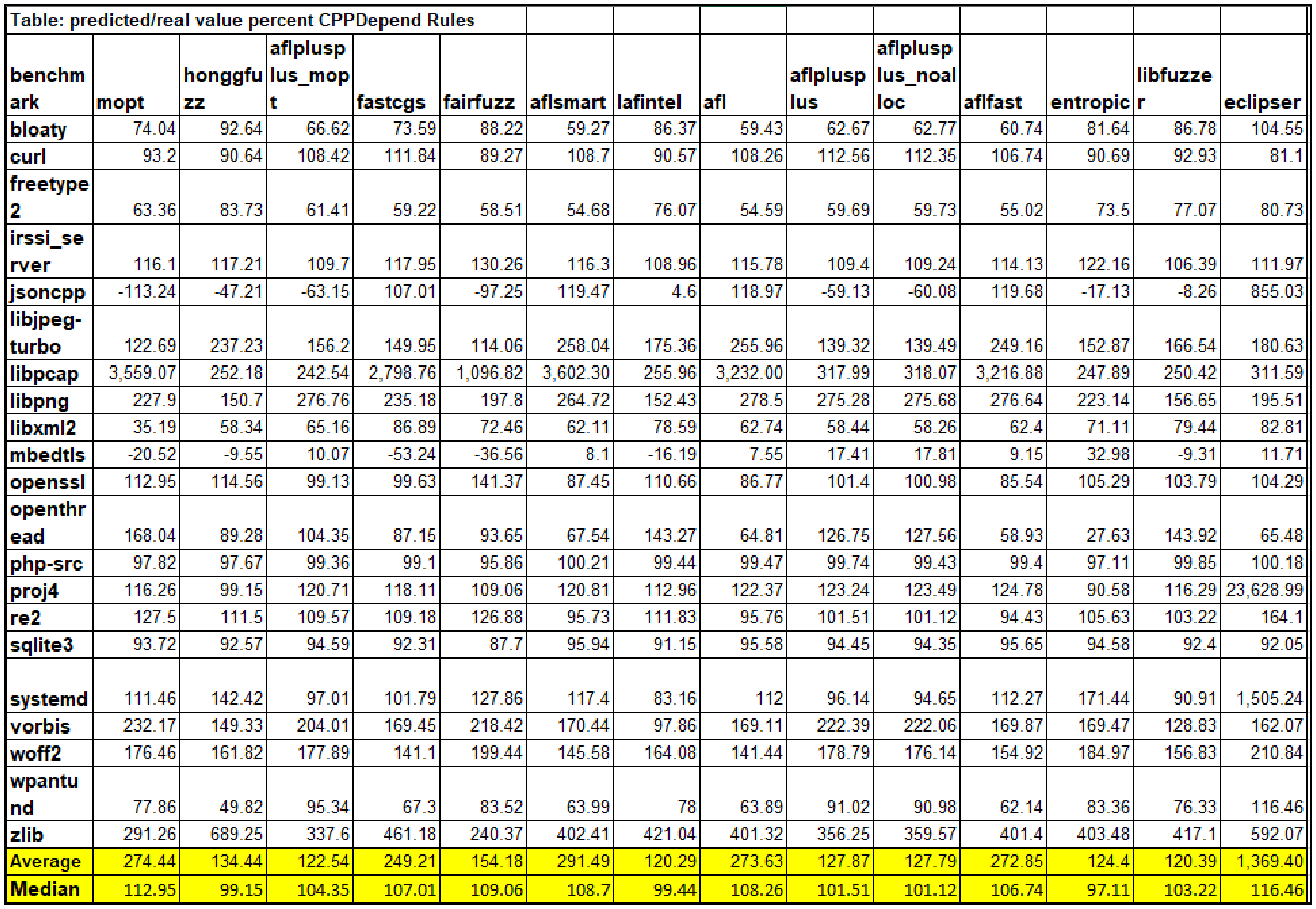


Table: predicted/real value percent CPPDepend Rules													
benchmark	mopt	honggfuzz	aflplusplus_mopt	fastcgs	fairfuzz	aflsmart	lafintel	afl	aflplusplus	aflplusplus_noalloc	aflfast	entropic	libfuzzer	eclipser
bloaty	74.04	92.64	66.62	73.59	88.22	59.27	86.37	59.43	62.67	62.77	60.74	81.64	86.78	104.55
curl	93.2	90.64	108.42	111.84	89.27	108.7	90.57	108.26	112.56	112.35	106.74	90.69	92.93	81.1
freetype2	63.36	83.73	61.41	59.22	58.51	54.68	76.07	54.59	59.69	59.73	55.02	73.5	77.07	80.73
irssi_server	116.1	117.21	109.7	117.95	130.26	116.3	108.96	115.78	109.4	109.24	114.13	122.16	106.39	111.97
jsoncpp	-113.24	-47.21	-63.15	107.01	-97.25	119.47	4.6	118.97	-59.13	-60.08	119.68	-17.13	-8.26	855.03
libjpeg-turbo	122.69	237.23	156.2	149.95	114.06	258.04	175.36	255.96	139.32	139.49	249.16	152.87	166.54	180.63
libpcap	3,559.07	252.18	242.54	2,798.76	1,096.82	3,602.30	255.96	3,232.00	317.99	318.07	3,216.88	247.89	250.42	311.59
libpng	227.9	150.7	276.76	235.18	197.8	264.72	152.43	278.5	275.28	275.68	276.64	223.14	156.65	195.51
libxml2	35.19	58.34	65.16	86.89	72.46	62.11	78.59	62.74	58.44	58.26	62.4	71.11	79.44	82.81
mbedtls	-20.52	-9.55	10.07	-53.24	-36.56	8.1	-16.19	7.55	17.41	17.81	9.15	32.98	-9.31	11.71
openssl	112.95	114.56	99.13	99.63	141.37	87.45	110.66	86.77	101.4	100.98	85.54	105.29	103.79	104.29
openthread	168.04	89.28	104.35	87.15	93.65	67.54	143.27	64.81	126.75	127.56	58.93	27.63	143.92	65.48
php-src	97.82	97.67	99.36	99.1	95.86	100.21	99.44	99.47	99.74	99.43	99.4	97.11	99.85	100.18
proj4	116.26	99.15	120.71	118.11	109.06	120.81	112.96	122.37	123.24	123.49	124.78	90.58	116.29	23,628.99
re2	127.5	111.5	109.57	109.18	126.88	95.73	111.83	95.76	101.51	101.12	94.43	105.63	103.22	164.1
sqlite3	93.72	92.57	94.59	92.31	87.7	95.94	91.15	95.58	94.45	94.35	95.65	94.58	92.4	92.05
systemd	111.46	142.42	97.01	101.79	127.86	117.4	83.16	112	96.14	94.65	112.27	171.44	90.91	1,505.24
vorbis	232.17	149.33	204.01	169.45	218.42	170.44	97.86	169.11	222.39	222.06	169.87	169.47	128.83	162.07
woff2	176.46	161.82	177.89	141.1	199.44	145.58	164.08	141.44	178.79	176.14	154.92	184.97	156.83	210.84
wpantund	77.86	49.82	95.34	67.3	83.52	63.99	78	63.89	91.02	90.98	62.14	83.36	76.33	116.46
zlib	291.26	689.25	337.6	461.18	240.37	402.41	421.04	401.32	356.25	359.57	401.4	403.48	417.1	592.07
Average	274.44	134.44	122.54	249.21	154.18	291.49	120.29	273.63	127.87	127.79	272.85	124.4	120.39	1,369.40
Median	112.95	99.15	104.35	107.01	109.06	108.7	99.44	108.26	101.51	101.12	106.74	97.11	103.22	116.46

VI: Problems Encountered

Several problems were encountered when gathering the data for the regression model and when creating the regression model. These problems can explain the inaccuracies and variability in predictions generated by this tool.

First, CPPDepend and most code analysis tools are not free, so there was little time to learn how to use CPPDepend and troubleshoot any problems with it during its free trial period. One problem that was not completely fixed was the discrepancy between operating system requirements for CPPDepend and the benchmarks. Most of the benchmarks require a Linux system to compile and run, while the version of CPPDepend used was for Windows. CPPDepend needs to know how to compile a program in order to analyze it. I was able to have CPPDepend compile all the benchmarks using Visual Studio, but every compilation had some errors. Most of the errors involved being unable to find some header files. The amount of compilation errors using Visual Studio project files was significantly lower than when using any other method, including creating a buildpath to the benchmarks inside a Windows Subsystem for Linux (WSL), MinGW, and manual build paths created using CMake.

The compilation errors can partially explain the differences in shared metrics data gathered using SourceMonitor and CPPDepend. For example, CPPDepend's count for Lines of Code was always smaller than SourceMonitor's count for Lines of Code. The other explanation for this discrepancy might be that CPPDepend ignores preprocessor lines because it analyzes the compiled code, while SourceMonitor only sees source code.

Second, Free code analysis tools like SourceMonitor do not gather as many program metrics as non-free analysis tools. With less data, the regression model is more likely to be highly variable. Indeed, the variance for SourceMonitor's regression model was very high, and this variance was reflected in the predicted performance values.

This project was also limited by the number of observations or benchmarks that had fuzzer performance data. Fuzzbench, the source for all the performance data in this project, only publishes performance data on about 20 different benchmarks. To get a more accurate regression model, there needs to be at least 50 different benchmarks or observations.

Finally, before creating a stepwise regression function, only obviously collinear variables were removed. Collinearity can introduce more variance and decrease precision and accuracy of the estimates in the regression model. To completely remove collinear variables, factor analysis should have been done on the variables as a whole.

VII: Conclusion and Future Work

As discussed, there are many fuzzers available, and new fuzzers are created every year. There is a growing effort to rank these fuzzers in order to find the best fuzzer, but until now, there has been no work in using program metrics to predict fuzzer performance. That is, there is no way to predict which fuzzer is best for a given program.

This project sought to rectify this and introduced a tool that predicts fuzzer performance given program metrics gathered using SourceMonitor or CPPDepend. The predictions are not perfect, and predicting using program metrics gathered using CPPDepend seems to generate the least variable and most accurate performances.

There is much to improve with this tool. The predictions are not the best because there simply was not enough data to create the regression model, so for future work, I can use Fuzzbench to gather more performance data on more benchmarks. I will need at least 50 benchmarks to create an accurate regression model. Next, I will eliminate the operating system discrepancy between the benchmarks and CPPDepend by simply using the Linux version of CPPDepend. This should make the data gathered using CPPDepend much more accurate. Finally, collinearity between program metrics will need to be eliminated before creating another regression model. As discussed, this would involve running a factor analysis on the program metrics gathered.

Once the problems encountered in this project are addressed, I can also expand this project to include more kinds of metrics, on top of those gathered using analysis tools. For example, there has been some work done showing that not all program metrics are equal when used for improving fuzzer performance [10,13]. Some of the metrics used to improve fuzzing performance are not ones I gathered, so it would be interesting to see if a regression model that included those metrics would be significantly more accurate.

Finally there was some work done analyzing how obfuscating a binary affected the accuracy of program metrics gathered [7]. Some fuzzers, specifically blackbox fuzzers, specialize in fuzzing binaries only. This project does not attempt to predict fuzzing performance of blackbox fuzzers, but it would be interesting to see how fuzzing predictions are affected when using an obfuscated binary.

References

1. A. Sepasmoghaddam and H. Rashidi, "A Novel Method to Measure Comprehensive Complexity of Software Based on the Metrics Statistical Model," 2010 Fourth UKSim European Symposium on Computer Modeling and Simulation, Pisa, 2010, pp. 520-525, doi: 10.1109/EMS.2010.92.
2. Ari Takanen; Jared D. Demott; Charles Miller (31 January 2018). *Fuzzing for Software Security Testing and Quality Assurance, Second Edition*. Artech House. p. 15. ISBN 978-1-63081-519-6. (archived September 19, 2018)
3. George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (*CCS '18*). Association for Computing Machinery, New York, NY, USA, 2123–2138. DOI:https://doi.org/10.1145/3243734.3243804
4. Google Fuzzbench: https://github.com/google/fuzzbench
5. Google Fuzzbench reports: https://www.fuzzbench.com/reports/index.html
6. Khan, Ali Athar, et al. "Comparison of Software Complexity Metrics." International Journal of Computing and Network Technology, vol. 4, no. 1, 2016, p. 19+. Accessed 9 Nov. 2020.
7. Kuznetsov, M A, and V O Surkov. “Analysis of Complexity Metrics of a Software Code for Obfuscating Transformations of an Executable Code.” *IOP Conference Series: Materials Science and Engineering*, vol. 155, 2016, p. 012008., doi:10.1088/1757-899x/155/1/012008.
8. N. Medeiros, N. Ivaki, P. Costa and M. Vieira, "Software Metrics as Indicators of Security Vulnerabilities," 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), Toulouse, 2017, pp. 216-227, doi: 10.1109/ISSRE.2017.11.
9. Patrice Godefroid. 2020. Fuzzing: hack, art, and science. *Commun. ACM* 63, 2 (February 2020), 70–76. DOI:https://doi.org/10.1145/3363824
10. Shudrak, Maksim O., and Vyacheslav V. Zolotarev. “Improving Fuzzing Using Software Complexity Metrics.” *Information Security and Cryptology - ICISC 2015 Lecture Notes in Computer Science*, 2016, pp. 246–261., doi:10.1007/978-3-319-30840-1_16.

11. SourceMonitor: http://www.campwoodsw.com/
12. T. M. Khoshgoftaar and J. C. Munson, "Predicting software development errors using software complexity metrics," in IEEE Journal on Selected Areas in Communications, vol. 8, no. 2, pp. 253-261, Feb. 1990, doi: 10.1109/49.46879.
13. Wang, Yanhao, et al. "Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization." *Proceedings 2020 Network and Distributed System Security Symposium*, 2020, doi:10.14722/ndss.2020.24422.

Appendix 1: Metrics Explanations

These metrics explanations are taken from the SourceMonitor help pages and the CPPDepend metrics explanation pages, respectively.

1a: Source Monitor Metrics

Statements: in C, computational statements are terminated with a semicolon character. Branches such as if, for, while, and goto are also counted as statements. Preprocessor directives #include, #define, and #undef are counted as statements. All other preprocessor directives are ignored. In addition all statements between an #else or #elif statement and its closing #endif statement are ignored, to eliminate fractured block structures. This metric counts all of the statements in a file, or in all files in a checkpoint.

Percent Branch Statements: Statements that cause a break in the sequential execution of statements are counted separately. These are the following: if, else, for, while, goto, break, continue, switch, case, default, and return. Note that do is not counted because it is always followed by a while, which is counted.

Percent Lines with Comments: The lines that contain comments, either C style (/*...*/) or C++ style (//...) are counted and compared to the total number of lines in the file to compute this metric. If you select the project option to ignore headers and footers (see create project, step 3), contiguous comment lines at the beginning and end of files are not counted as comments.

Functions: Number of functions found in a file, or in all files in a checkpoint.

Average Statements per Function: The total number of statements found inside of functions found in a file or checkpoint divided by the number of functions found in the file or checkpoint.

Maximum Function Complexity: The complexity value of the most complex function in a source code file.

Maximum Block Depth: Maximum Block Depth is the maximum nested block depth level found. At the start of each file the block level is zero. Depths up to 9 are recorded and all statements at deeper levels are counted as depth 9. This is indicated by the "9+" label for the deepest level. The maximum depth displayed in the Project View and the Checkpoint View will show the actual maximum depth (instead of "9+") if the language option Display Measured Maximum Block Depth is checked in the Options dialog tab for a language. This actual maximum depth is capped at 32.

Average Block Depth: Average Block Depth is the average nested block depth weighted by depth. While nesting can be used alone in most languages, nested blocks are almost always introduced with execution control statements such as “if”, "case". and “while”. As the depth grows, the code gets harder to read because with each new nested depth level, more conditions must be evaluated if you want to know when the code will be executed. Average block depth is the weighted average of the block depth of all statements in a file or checkpoint.

Average Complexity: The Average Complexity metric is a measure of the overall complexity measured for each method (and, if present, each function) in a file or checkpoint. It is computed as a simple arithmetic average of all complexity values measured for a file or checkpoint.

1b: CPPDepend Metrics

There are too many CPPDepend metrics to list, but all of them can be found here https://www.cppdepend.com/metrics.