Percentages, Probabilities and Professions of Performance

Jim Alves-Foss
Center for Secure and Dependable Systems
University of Idaho

Abstract

Experimental cybersecurity publication should provide readers with a reliable report of the experimental methods, dataset(s) used and full analysis of the results to allow the readers to fully understand the capabilities and limitations of the experiment, and to compare the results to other similar tools or processes. This paper provides an example of looking at experimental results a few different ways, in an attempt to get a better understanding of the underlying processes. We encourage other authors to do the same. We conclude with some basic recommendations.

1 Introduction

How often have you read a cybersecurity paper that makes a claim such as "We achieve a 95% True Positive rate" or "We ran our code on 1,000 test cases and our tool's outperforms our competitors 98% ($p < 0.05$) of the time"? What do these values mean? Are they even useful to the reader? This paper highlights some recent examples of research that leaves unanswered questions. Although we do not propose a precise set of guidelines, we request that conference committees develop and publish guidelines for authors requiring more detailed analysis of results and that reviewers reject papers that do not meet these standards.

Note that this paper does not address cybersecurity publications that introduce a single instance of an attack or vulnerability. These vulnerability reports are useful learning exercises, and can help demonstrate the vulnerabilities in commercial systems. Instead, this paper is focused on structured experiments where a tool or process is applied to a dataset, with the implicit assumption that the experiment is repeatable or comparable to other results. For this to hold, the experimental method, metrics applied, and results, must be presented in a clear and complete manner.

2 What does the *mean* mean?

Many experimental cybersecurity publications will present an average of some metric related to the author's work. In most cases, a good number is needed for a positive review and subsequent publication. However, too often, there is not enough information for the reader to understand the analysis; they only get enough information to understand that the author is claiming success.

A *mean* is an average of a metric applied to the dataset. Although there are different types of means, including geometric mean and harmonic mean; most authors use the arithmetic mean. And in many publications, we see it as the only statistic presented. However, a mean does not describe how well the dataset samples the population. It does not describe the statistical power of the result, in other words, it does not tell the reader if there are enough samples for the statistics to be meaningful. Is 30 test cases enough, or do we need 300, or 3000? If the 30 test cases cover a wide range of features related to what is being studied compared to 3000 very similar test cases, the 30 may be more useful.

2.1 Example from past work

This recently came to light with an experiment we were running. The precise details and our results are not relevant to this paper; it is sufficient to summarize the experiment. We were looking at the ability to detect function start addresses from fully stripped binaries, a non trivial problem. We read a couple of papers published in top conferences. In the first work we read, from Bao et al. [3], results for their tool Byteweight were presented as in Table 1. In a subsequent paper by Andriesse et al. [1], results for their tool Nucleus were presented in as in Table 2, as well as using graphs.

Bao et al. used a Unix utility dataset consisting of binutils, coreutils and findutils packages, compiled with four different optimization options, for x86 and x64 architectures and with two different compilers, gcc and icc[1]. The goal was to evalu-

[1]Both Bao et al. and Andiresse et al. also looked at Windows applications

Table 1: Precision Recall of function start identification (reproduction of Table 2 from Bao et al. [3])

	GCC			ICC		
	Precision	Recall	Time(sec)	Precision	Recall	Time(sec)
Rosenblum et al. [7]	0.4909	0.4312	1172.41	0.6080	0.6749	2178.14
BYTEWEIGHT (3)	0.9103	0.8711	1417.51	0.8948	0.8592	1905.34
BYTEWEIGHT (no-norm)	0.9877	0.9302	19994.18	0.9727	0.9132	20894.45
BYTEWEIGHT	0.9726	0.9599	1468.75	0.9725	0.9800	1927.90

Table 2: Precision Recall of function start identification (reproduction of Table1(a) from Andriesse et al. [1])

	gcc x86	gcc x64	clang x86	clang x64	VS x86	VS x64
IDA Pro 6.7	0.98/0.78	0.97/0.74	0.98/0.78	0.98/0.77	0.84/0.93	1.00/0.94
BAP/ByteWeight 0.9.9	0.68/0.83	0.70/0.66	0.52/0.71	0.73/0.49	0.63/0.74	0.69/0.56
Dyninst 9.1.0	0.93/0.91	0.96/0.74	0.98/0.95	0.88/0.72	–	–
Nucleus	0.98/0.96	0.98/0.96	0.96/0.97	0.96/0.95	0.86/0.96	0.95/0.94
△Nucleus	+0.00/+0.05	+0.01/+0.22	−0.02/ +0.02	−0.02/+0.18	+0.02/+0.03	−0.05/+0.00

ate if machine learning could be used to recognize and detect function boundaries. Andriesse et al. used the same Unix utility dataset, along with SPEC CPU2006 and a dataset consisting of 5 server applications. Andriesse et al. also compiled with 2 compilers (gcc and clang) and four optimization levels. In both cases, the assumption is that difference introduced by optimization level and compilers will generate different binary patterns for function starts[2].

One benefit of this prior work is that Bao et al. made their tool and dataset available [2] and Andiressee et al. made their code available [9]. Byteweight uses a machine learning approach which takes over 500 hours and over 500 GB of disk space for the training phase. Nucleus processes the whole dataset in under 20 minutes.

2.2 Percentages in the past work

As part of our research we ran Nucleus on the Unix utility dataset, using gcc and icc versions from the Bao et al. website [2], and compiling our own versions with Clang. We also built SPEC CPU 2017 benchmarks (including Fortran, C and C++ code) for x86 and x64 using gcc, icc and clang[3].

If we present our results using averages, as the source papers did, we get Table 3. A detailed breakdown like this is usually the best we get in a paper. The breakdown allows the reader to determine that there were issues with the SPEC CPU 2017 dataset, and with optimization O1 for Unix utilities and the authors feel that their work is done.

and Visual Studio compiler, but those are not addressed in this summary.

[2]Andriesse et al. did address concerns with the Unix utility dataset, stating that "we found that it contains many binaries with large amounts of common functions." Therefore they also used SPEC CPU 2006 C and C++ benchmarks along with 5 popular server applications to provide diversity to their dataset.

[3]Some of the SPEC CPU 2017 benchmarks work only for x64.

Table 3: Average F1 values of different subsets of combined datasets, clang compiler, x64.

Data Set	F1 Func. Start Identification
Combined Data Sets, all optimizations	95.15%
SPEC CPU 2017, all optimizations	85.98%
Unix Utilities, all optimizations	97.24%
Unix Utilities, optimization O0	99.63%
Unix Utilities, optimization O1	94.27%
Unix Utilities, optimization O2	97.53%
Unix Utilities, optimization O3	97.51%

2.3 Looking at the data

One way of examining data is to look at it graphically. We took the results from our experiments with Nucleus and did just that. First we plotted a histogram of the F1 values for the combined datasets (Fig. 1). This tells us there is a heavy bias towards good performance. It also shows us that the distribution is not normal, so we will have to be careful if we wish to compare these results with other tools. Just comparing averages will probably not be enough, and we can not use t-test or other statistical tools that require a normal distribution in the population. Most cybersecurity experiments will not have a normal distribution, or may use insufficient sample size for the type of statistical analysis they present. Those that still present a comparison using satistical test, may try to bolster their claims with a p-value (such as $p < 0.05$). However, this is misreported, since the test is not valid on the data.

Next we plotted the data using a scatter plot (Fig. 2). We plotted the SPEC CPU 2017 data first and then the Unix tools, grouped by name. This grouping is a byproduct of the way Bao et al. named the executable and sorting them alphabetically. A sample name was `clang_binutils_64_01_strings`, specifying compiler, toolset, architecture, optimization level and individual tool name. This scatterplot reveals the difficultly with SPEC CPU 2017 dataset. It also shows a low clustering artifact. Fortunately with Excel we can hover over a datapoint and see which item is plotted. This allowed us to see that there was a clustering for optimization O1 among the coreutils.

Given this information, we modified the scatterplot to just show the Unix utilities, now grouped by optimization level and then by name (Fig. 3), and zoomed in. We can see the drop in F1 values with increased optimization, and the artifact from coreutils. This tells the reader there is something special here, and these values dragged down the overall average.

Since the results are based on percentage of function starts found, and some of these utilities are small, we wondered if a binary with a small number of functions would necessarily have a lower F1 value. The binaries ranged from 16 to almost 50,000 functions. We regrouped the data, still grouping by optimization levels, and then sorting within the level by the number of functions (Fig. 4). The results here are very interesting. First in all cases we see a swoop up as the number of functions increases, validating our opinion that a few missed functions could drive the average down, and as the number of functions increased, the overall performance improved. Fig. 4 also clearly highlights the strange artifact in the O1 optimization level. A further look at the raw data shows that these binaries have 20-25 functions that are missed by Nucleus. This is probably a set of common shared functions, compiled in a way that violates the assumptions in Nucleus's analysis.

This way of presenting and looking at the data, gives the readers a clue to the limitations of the presented work, and a hint to future work and exploration.

3 How accurate is accuracy?

In an experimental cybersecurity paper, we sometimes see the word "accuracy". If a paper is evaluating an intrusion detection technique, it has evaluations of events, resulting in the typical confusion matrix (Table 4). Authors can calculate a lot of different statics by combining the values from this matrix in different ways. Some of the most common are presented in Fig. 5. Here *true positive/negative* indicates that the tool correctly identified a condition and **condition positive/negative** reflects reality.

It is important to note that accuracy counts all the true positives and true negatives. If most of our events are benign, the true negatives will outweigh anything else, and you will have very high accuracy. For example, assume there are two intrusion attempts out of a dataset of 1,000 connections. If your tool reports two intrusions, one accurately and one inaccurately (for one true positive, one false positive and one false negative) you have 998/1000 = 99.8% accuracy. However, your precision and recall are 50%, and therefore so is your F1

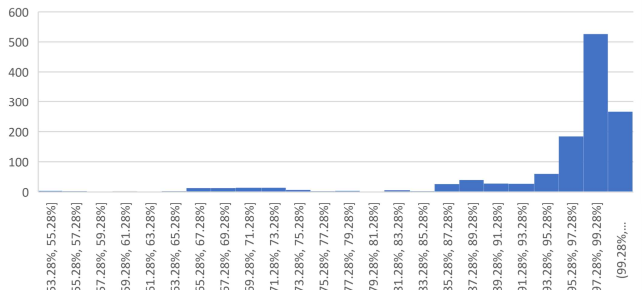

Figure 1: Histogram F1 values for Nucleus on combined datasets, clang compiler, x64.

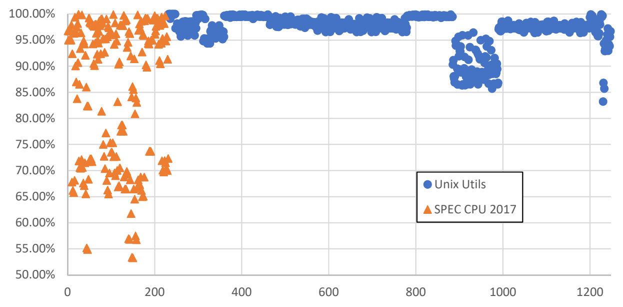

Figure 2: Scatterplot of F1 values for Nucleus on combined datasets, alphabetical within datasets, clang compiler, x64.

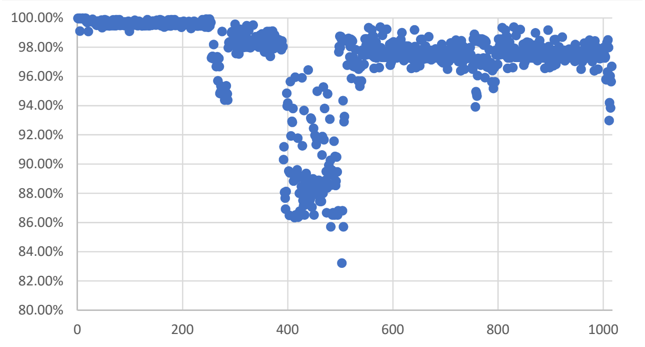

Figure 3: Scatterplot of F1 values for Nucleus on Unix utility dataset, group by optimization levels, clang compiler, x64.

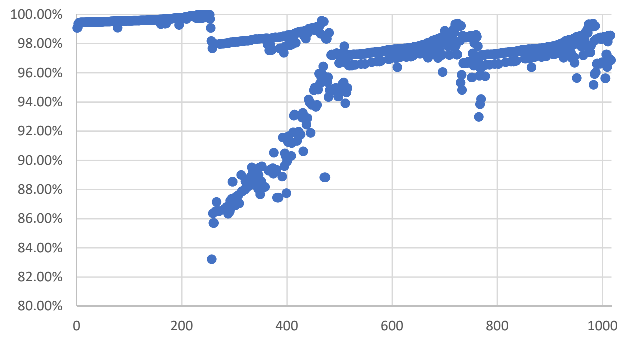

Figure 4: Scatterplot of F1 values for Nucleus on Unix utility dataset, group by optimization levels, then sorted by number of functions, clang compiler, x64.

Table 4: 2×2 confusion matrix

	Ground Truth	
	Condition Positive	Condition Negative
Predicted Positive	True Positive	False Positive Type 1 error
Predicted Negative	False Negative Type 2 error	True Negative

value. If you reported just one intrusion correctly (one true positive and one false negative), you have 99.9% accuracy, 100% precision and 50% recall for a 75% F1 value. Lastly, if you only have one false negative and no true positives, you would have a 99.7% accuracy but 0% precision, recall and F1 values; not useful, but highly accurate.

4 Recommendations

There are several ways to address the issues raised in this paper. The most important is to look at the data and understand its structure. The second is to understand the type of analysis you are trying to perform and then use the appropriate statistics – this may involve speaking with a statistician.

4.1 Comparing Approaches

In computer science it is easy to reuse the same dataset for multiple tests. Unlike experiments in the *real world*, we will not have to address learning, boredom or other order effects (the impact of applying test A first on the results of a subsequent test B). When you are comparing two different tools or techniques to the same dataset, you can use a dependent (paired) statistical test. However, simply search the Internet

$$\begin{aligned}
Prevalence &= \frac{\Sigma \text{ Condition positive}}{\Sigma \text{ Total Population}} \\
Accuracy &= \frac{\Sigma \text{ True positive} + \Sigma \text{ True negative}}{\Sigma \text{ Total Population}} \\
Precision &= \frac{\Sigma \text{ True positive}}{\Sigma \text{ Predicted positive}} \\
Recall &= \frac{\Sigma \text{ True positive}}{\Sigma \text{ Condition positive}} \\
F1 &= 2 \times \frac{Precision \times Recall}{Precision + Recall}
\end{aligned}$$

Figure 5: Some definitions from the confusion matrix.

or drop-down menu of your statistical software for the appropriate test is not enough.

- **Distribution Assumptions**. Statistical tests are designed with specific assumptions regarding distributions of data. The common *t-test* and the *dependent t-test* assume the data has a normal distribution which is often violated in experiments (see Fig. 1). The *Wilcoxon-signed-rank test*, also used for comparing paired results, assumes that the set of differences is symmetrical in shape [6], whereas the *Sign test* does not [5]. The *Kruska-Wallis test* always works with pairs, but assumes one results stochastically dominates the other [10]. We recommend the *Sign test* for the type of analysis we discuss in this paper.

- **Power**. Statistical tests will often be used to test the null-hypothesis (that there is no difference between two groups), and results are often reported with a probability (95% or 99%). However, the results should be presented with a confidence interval. For example: Tool A outperforms Tool B by $3 \pm 1\%$ (p=0.05) tells the reader that there is a 95% confidence that the difference between the tools is between 2% and 4%. There are tools to calculate the confidence interval, which is dependent upon the size of the dataset and the overall population. Or, you can use a tool to determine the sample size you need for the confidence interval you want [4]. These tools give a conservative estimate, but give you idea of the problem researchers face. For the preceding example, to get 95% confidence with a 1% confidence interval, assuming a population of 1 million software programs, requires 9,513 test cases. Even a 3% interval (in the above case, the difference would be from 0 to 5%) requires 1,066 test cases.

- **Randomness**. Samples need to be drawn randomly, or tests need to be performed to show that the sample is an accurate representation of the whole population. Many statistical tests assume randomness to allow for generalizing results to the full population.

4.2 Reporting Accuracy

It is important to understand the data. If you are analyzing a detection tool, you need to understand the population.

- **Percentage of True Positives**. Most cybersecurity tools are looking for relatively rare events out of a population. For example, vulnerabilities in code, attempted DDOS attacks, or malicious network packets. Therefore it is important to report the correctness in detecting the true positives and not the true negatives. The exception may be in detecting email spam, which accounted for over 55% of email traffic in January through March 2019 [8].

- **Sample size**. Again sample size is very important when the prevalence of a true positive is low. With a small dataset, you may have only a few samples with condition positive, and your highly correct tool may miss a few making the results look bad, or a poor tool may get lucky making the results look better than they are. Here again, confidence intervals are useful.

5 Conclusion

We have performed a cursory review of some of the recent top cybersecurity conferences. We have seen accepted papers that do not fully describe or justify their datasets, ones that present just averages, and those that report useless accuracy or misuse the word accuracy.

In this paper we have shown how data from some real experiments can hide information that would be valuable to the readers, helping them better understand the strengths and limitations of the work. We request that review committees and reviewers push authors to provide a better description of their analysis. To summarize:

- Use a dependent (or paired) statistical test such as the *Sign test* if you are comparing results of different tools on the same dataset.

- Use a large dataset of randomly selected test cases.

- Report the confidence interval of your results – and assume a very large population if you are testing software.

- Consult a statistician to make sure your experiments is designed correctly.

- When reporting how well a detection tool performs, pay attention to the ratio of true positives to true negatives. If the population has a small percentage of true positives, use F1 values or some other report of the tools correctness of reporting the true positives.

Lastly, we want to state that our use of the Bao et al. [3] and Andriesse et al. [1] works was meant to be illustrative and not critical of the quality or value of their work. We are especially grateful for these authors making their tools publicly available, a practice that helps the community and should be strongly encouraged by conference committees.

References

[1] Andriesse, D., Slowinska, A., and Bos, H. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)* (2017).

[2] Bao, T., and Brumley, D. Byteweight: Recognizing functions in binary code. `http://security.ece.cmu.edu/byteweight/`.

[3] Bao, T., Burket, J., Woa, M., Turner, R., and Brumley, D. Byteweight: Learning to recognize functions in binary code. In *Proc. USENIX Security Symposium* (2014), pp. 845–860.

[4] Creative Research Systems. Sample size calculator. `https://www.surveysystem.com/sscalc.htm`.

[5] Laerd Statistics. Sign test using SPSS statistics. `https://statistics.laerd.com/spss-tutorials/sign-test-using-spss-statistics.php`.

[6] Laerd Statistics. Wilcoxon signed-rank test using SPSS statistics. `https://statistics.laerd.com/spss-tutorials/wilcoxon-signed-rank-test-using-spss-statistics.php`.

[7] Rosenblum, N. E., Zhu, X., Miller, B. P., and Hunt, K. Learning to analyze binary computer code. In *National Conference on Artificial Intelligence* (2008), pp. 798—-804.

[8] Statista. Global spam volume as percentage of total e-mail traffic from january 2014 to march 2019, by month. `https://www.statista.com/statistics/420391/spam-email-traffic-share/`.

[9] VUSec. Nucleus source code, 2018. `https://www.vusec.net/projects/function-detection`.

[10] Wikipedia. Kruskal–wallis one-way analysis of variance. `https://en.wikipedia.org/wiki/Kruskal%E2%80%93Wallis_one-way_analysis_of_variance`.