Work In Progress: Optimizing Data Mediation Placement with Pareto Analysis

J. Peter Brady
Computer Science Department
Dartmouth College
Hanover, NH USA
jpb@cs.dartmouth.edu

Sean W. Smith
Computer Science Department
Dartmouth College
Hanover, NH USA
sws@cs.dartmouth.edu

***Abstract*—We demonstrate a novel way of using data mediation and Pareto optimization to add Language-Theoretic security to software applications by evaluating the application's data structures and their interaction with other data through the application's operation. We create a general model and automated programs that provides a developer with a straightforward way to select a balance point between security coverage and performance.**

I. Introduction

Data is an essential part of today's computer systems; without it, there would be no high-speed communications for connecting businesses, factories, homes, or other vital devices, nor would there be complex media such as video streaming. Rapid changes in technology have accelerated the need for bandwidth; for example, North America currently contributes about "38% of interconnection bandwidth and is expected to grow by a 46% compound annual growth rate" [1].

Quickly moving vast amounts of data means system performance is a critical consideration in computing systems, but our data is only as good as its integrity. Data integrity is one facet of security – ensuring that data conform to the design specifications, that malicious or thoughtless modifications cannot occur, and that applications can only access the data needed at that time.

We can increase security in an application by applying the tenets of Language-theoretic Security (LangSec) [2] principles to prevent malicious behaviors. LangSec is a "design and programming philosophy that focuses on formally correct and verifiable input handling throughout all phases of the software development life cycle." [3]

If we think of a routine in a program with input data, the LangSec philosophy has the developer describe all the valid inputs for each input variable as elements in a formal language. The input description is then transformed into a recognizer inserted into the routine. The recognizer can then pass conforming, structured data to the processing code while rejecting data that does not conform.

One goal of LangSec is to provide a verifiable recognizer used everywhere; this stops developers from inserting their own input checking code that may contain *ad-hoc* parsing bugs and provide an exploitation path. Input processing bugs are the largest class of failure/exploit mechanism, so it is a critical hole to close. Another goal is to separate input data parsing from the computation logic, allowing only formally correct data to be processed. The processing code only has access to the checked data, removing possible exploits. In this way LangSec performs as a mediation statement, preventing illegal data flows between input processing and data processing.

Performance and security end up opposite to each other – having the best performance dictates that we make the system operate as quickly as possible, while constructing the best data security using an approach like LangSec puts extra tests or checks in place, adding to the amount of code and potentially slowing the system down.

Dans ses écrits, un sage Italien
Dit que le mieux est l'ennemi du bien[1].

–Voltaire, La Bégueule [4]

The balance point comes down to cost – either the cost of performance lost for security or the upfront cost of paying to add security to an application to avoid the more considerable downstream cost of repairs or lawsuits. Without having a cost methodology, increasing performance or adding more security to an application might not be justified. Marketers and economists think in terms of a price-performance or cost-benefit ratio for a product, where products with a lower price-performance ratio are desirable; many startup companies fail to keep this ratio in mind [5]. Our task is to find a methodology for creating a unified cost-benefit model to measure performance versus security that works for many programs. We need to narrow our scope to the most used data in an application – rather than trying for an insurmountable "best", we can search for the "good" changes and get a security boost without a huge performance penalty.

II. Obtaining Balance with Pareto Analysis

One way to create a model to balance performance and security is to narrow our scope of security modifications to the

[1]Translation: In his writings, a wise Italian said that the best is the enemy of the good.

most used functions in an application; rather than trying for a difficult-to-reach "best", we can search for a point where we have good performance and security goals.

The method we use to search for "good" rather than "best" is based on the Pareto Principle, which states that "in any population that contributes to a common effect, a relative few of the contributors – the vital few – account for the bulk of the effect" [6] (see the Appendix for additional background).

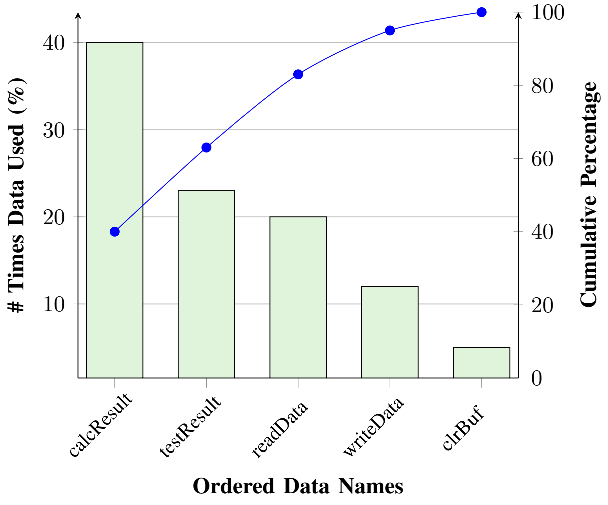

Fig. 1. Sample test results for a program's data displayed in an ordered Pareto histogram.

We can look at the Principle as an optimizer – can LangSec modifications to the most used data structures give us nearly the same results as mediation of all the data? More importantly, can we select the amount of performance degradation we would accept and see the amount of security coverage we receive?

The following describes the use of the Pareto Principle to create a process that will allow us to calculate the amount of LangSec modification necessary for the target application to have a specified level of coverage and a specified amount of performance. The development specifics are discussed in Section III.

A. Create a Pareto Histogram

We define a Pareto histogram where we rank the qualifying data structures in an application in order by the number of times called. A qualified data structure is global to the program or used by one or more subroutines and can be modified by any of those routines.

For example, Figure 1 shows a hypothetical program where we counted the number of times the data structures were accessed. We collect the number of times accessed, order the data from most to least accessed, and plot the results. We can see from the cumulative percentage that the *calcResult*, *testResult*, and *readData* data structures are in the Pareto "vital few" and account for 83% of accesses to the data structures.

B. Evaluate Thresholds Along the Pareto Histogram with Data Mediation

We can now add LangSec modifications to the routines by using the created Pareto histogram to add parsing in order from the most-used data structures to the least-used. We define a variable threshold called τ where $0 \geq \tau \geq 100$. $\tau == 0$ is the baseline application with no modifications while $\tau == 100$ is the application with every qualifying data structure LangSec modified. Using our example in Figure 1, we can test at each data element or $\tau = \{0, 40, 63, 83, 95, 100\}$.

a) Data Mediation: A graph can represent a program's structure; for example, each vertex in the graph is a subroutine, and the edges the connections between specific subroutines. Mediation uses the graph-cut, a function to separate selected vertices into two subsets, to separate a source and sink vertex, with a transformation function inserted between source and sink.

There are many examples of mediation in the literature; [7] and [8] create access control policies that prevent illegal information flow. A similar mediation system [9] looks at resolving type errors in a program by automatically adding mediation statements. They construct a graph-cut solution to describe the information flow where source and sink nodes define security boundaries. A minimum cut of the graph describes the least number of mediation statements needed to type check. A mediation system for placement of security mechanisms such as Intrusion Detection Systems (IDS) in a network is described in [10]. Finally, a messaging system uses mediation to modify message flows by finding the optimal placement for replication of stateless message transformations, allowing for multiple brokers to divide the workload [11].

Our mediation system builds a data-flow graph that shows where data reacts with the program's routines and whether those routines modify the data. We flag the data inputs and outputs and use them as sources and sinks, respectively, to calculate the graph's vertex-cut, separating inputs from outputs. The mediator's initial run (i.e., $\tau == 0$) also collects the usage counts for each piece of data for use in the Pareto analysis.

Once done with data mediation, we build the parsing solution – a new formal language description of inputs to feed to the parser/combinator. We use the data description to LangSec modify only the data structured described by the selected τ; for example, if $\tau == 10$, then we modify the top 10% of the data structures. We then repeat the modifications for each τ we want to model.

After completing the modifications, we compile and run the application to obtain its speed of execution. The input is a known set of fuzzer data to collect the number of errors that occur during the run. We convert the values to a normalized form against the original, unmodified version of the program. A flowchart is shown in Figure 2.

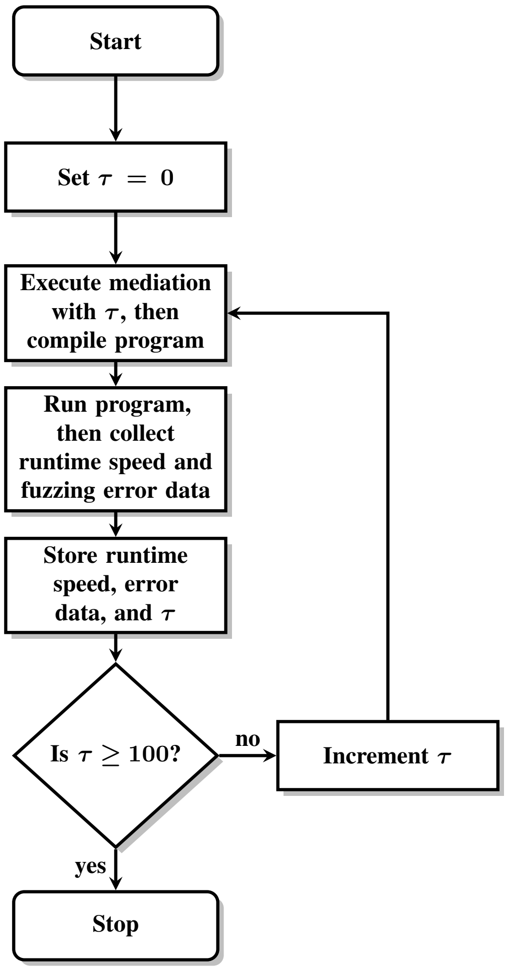
Start
Set $\tau = 0$
Execute mediation with τ, then compile program
Run program, then collect runtime speed and fuzzing error data
Store runtime speed, error data, and τ
Is $\tau \geq 100$?
no
Increment τ
yes
Stop

Fig. 2. Data collection flow chart for application training data

C. Plot Application Execution Speed and Data Errors

If we evaluate thresholds over many training applications, we can plot each in a scatter graph of application speed versus fuzzer data errors. By overlaying the data from each test application, we hope to build a generalized graph, which we can then use to select a level of performance and receive a corresponding level of data security, or vice versa.

D. Classify the Data

When we build our graph, we also overlay τ at that point. With enough data points, we can use the collected τ from our sample collection to create a mesh graph using a non-parametric machine learning technique such as k-Nearest Neighbors (k-NN) [12]. From this graph, we can predict τ for a new application given the error rate or the speed degradation desired.

E. Verify the Data

Once we have a correlated model, we need to verify that it truly works. The first step takes some of the training programs we used to build the model, selects an error rate and speed degradation, uses the returned τ to build the application, then checks to see how close the actual values track our input values.

The second step is taking a new application, building several versions with different error rates and speed reductions, and validating the numbers against the predicted data from the model.

III. Development Model

As mentioned in Chapter II, we can balance performance and security by narrowing our scope to the most used data by Pareto analysis, select our performance or security target, run our mediation system with the target input, then compare processing speed and code coverage against our initial mediation system model. The primary engine of our work is the data mediator.

A. Data Mediation

Our mediation system traces the data structures through a program, using graph-cuts to select the optimal points for adding LangSec parsing. We display a block diagram of the complete data mediation system in Figure 3 and annotate each step in red on the diagram.

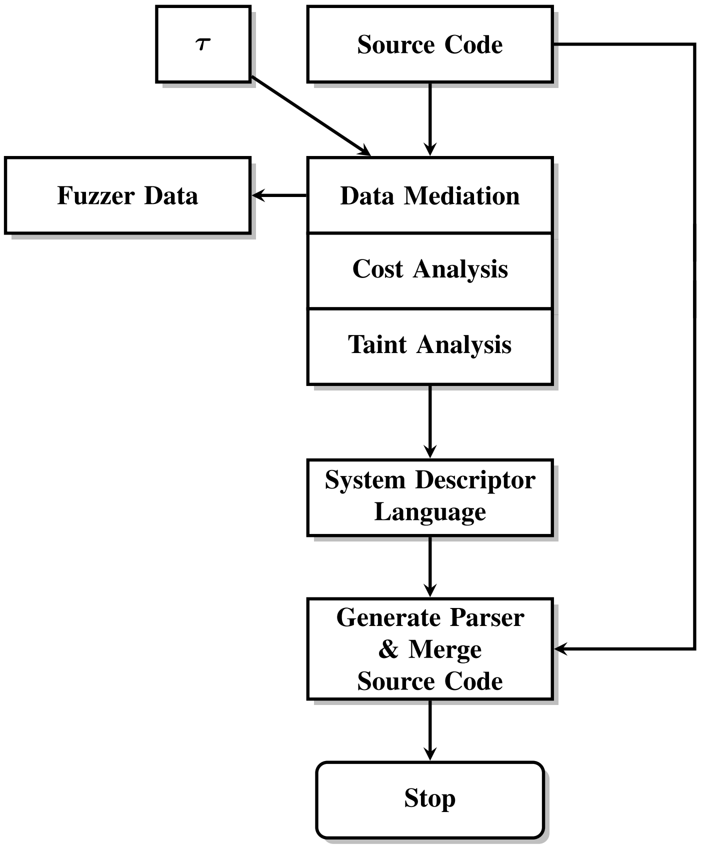

Fig. 3. Data Mediation System processing block chart.

Well defined functional areas allow us to have an API between functions, allowing flexibility if we decide to change or update a function with better technology.

a) Mediation: Program source code is read into the mediator, building a graph to find the optimal placement for data checking. If not specified, the mediator will collect and output all data found; however, if $\tau > 0$ is input at this stage, only the top τ percent of the most-used data structures will be passed in the graph.

Upon completing the initial graph-cut, the mediator looks for optimizations by first sending the data for cost analysis and then to a taint analyzer.

The next two steps are optimizations to the data graph. We will construct the system to be able to turn either optimization off to verify effectiveness. The cost analysis optimizer uses narrow-focused refactoring of the mediation graph to look for any time wasted in parsing unused data; for example, a routine that uses only one element in a structure only needs that element validated with the rest of the structure ignored and invalidated. The analyzer returns the results of any pruning to the mediator.

The taint analysis optimizer runs a fast static analysis to quickly look for other data interactions that may have been missed by the mediator. We initially expect to use this only for our initial data collection to verify the operation of the mediator.

While we recognize that many programs already contain shotgun parsers – code that intermingles input recognition with input processing – we will not attempt to remove those patterns in our initial build. We are investigating the possibility of using static analysis results to remove this code by methods similar to those described in [13] in a future version.

b) Fuzzer Data: The mediation engine drives an evolutionary fuzzer that has some awareness of the internal data patterns. The generated fuzzing patterns will be used during the validation step to create a metric for how well the mediation system worked.

c) System Descriptor Language: We convert the mediator's graph output into a Source Descriptor Language (SDL). Using an SDL allows us to provide an abstraction between the mediator and parser systems, allowing us the flexibility to try different parser/combinators.

d) Generate Parser and Merge Source Code: The generated SDL drives the generation of the parser/combinator code. We use the position information determined during mediation to merge the parser code into the source code.

B. Mediation System Testing and Validation

We need to validate our code coverage's completeness once we run the target source code through the mediation system. One way is to run a dynamic taint analysis against the compiled test programs. We will use an analysis framework such as Dytan [14] along with the fuzzer data generated by the mediator.

C. Pareto Analysis and Machine Learning

We will use standard Linux applications that can be run from the command line to provide the model's training set. We will build them with the mediator for each of the model τ set, run with known-good input data to get the base execution time, then run against a model fuzzer with the specific fuzzer data for that application to collect the errors.

Once we complete the training set applications, we can build the initial model with our created Python apps to take the set of error rate, time degradation, and τ values from each application as input. It will first extrapolate a curve from the scatter plot data of time versus error, then create a mesh of τ as a classifier using k-NN. We will initially start with a small τ set ($\tau = \{0, 25, 50, 75, 100\}$) to see how well the data correlates; if we need a higher level of granularity, we can make the τ increment smaller.

Once there is enough data to correlate, we will have a Python-based "solver" that can take two values (error, time, or τ) and return the third. We should also be able to enter either error or time and have the solver return the other value plus the τ classifier.

When we test previously unseen programs, this data will help calculate deviation from the curve. We expect to occasionally create a full set of data for programs that deviate significantly from the curve and add that data set as new learning data to improve our solutions. Initially, we are using $\pm 5\%$ for either error rate or speed degradation as the tolerance level. Figure 4 summarizes the process as a flow chart.

IV. Discussion and Next Steps

The initial step is testing our thesis by hand-tracing the data structures through some C-based applications used to train the model, then creating a Pareto histogram of the data. From the histogram, we will modify each program using a τ increment of 25 ($\tau = \{0, 25, 50, 75, 100\}$). Since we plan on using C programs for this test, we will use Hammer [15] as the parser/combinator. We are currently selecting the initial sample programs and expect to have our first data points soon.

Once we get a select set of samples, we expect to end up with a quadratic-derived plot something like the one in Figure 5. The normalized baseline data (i.e., with no modifications) will be at (1,1) on the plot. We expect that the actual data will look like an inverse of the cumulative percent line in the Pareto chart. The most significant drop will be the most-used data structures and leveling out to an asymptotic value greater than zero as τ approaches 100%.

As we accumulate data, we can create a trend line (the red dashed line in Figure 5). We can use this mathematical approximation to act as a solver. We can enter the error rate or execution speed and obtain the other value, allowing us to test the model's accuracy as we experiment with the complete mediation system.

When we plot the data, we also keep track of the τ value for each point. We display this in Figure 6, grouping the τ values into five buckets.

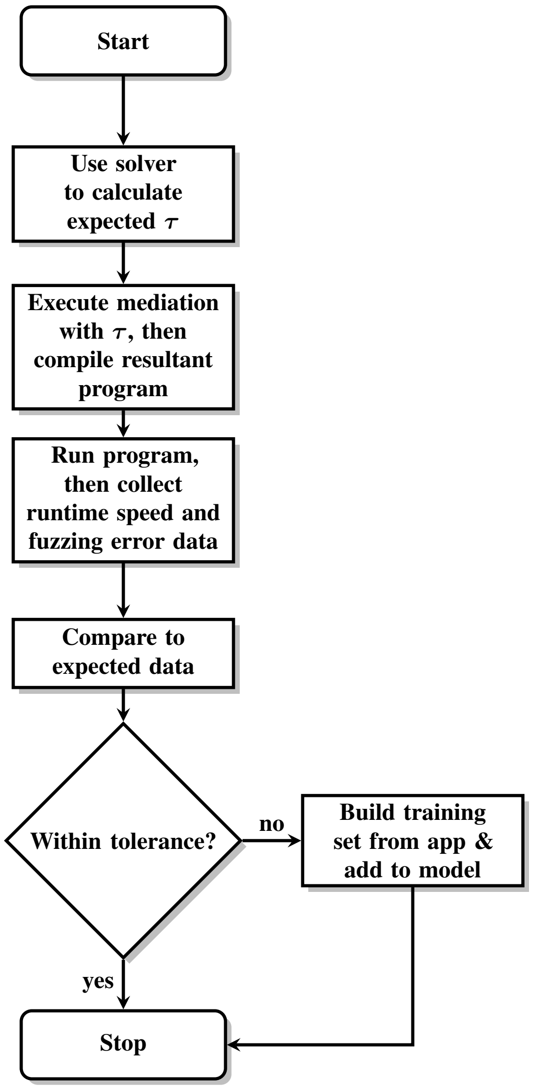
Start
Use solver to calculate expected τ
Execute mediation with τ, then compile resultant program
Run program, then collect runtime speed and fuzzing error data
Compare to expected data
Within tolerance?
no
Build training set from app & add to model
yes
Stop

Fig. 4. Flow chart for mediating new applications from the collected training data.

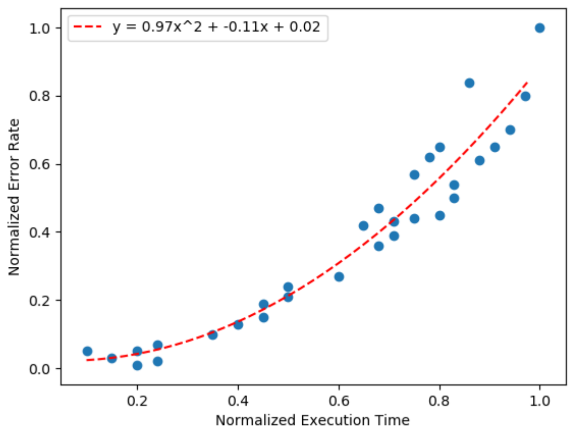

Fig. 5. Sample Normalized Time versus Error Rate with Trend Line.

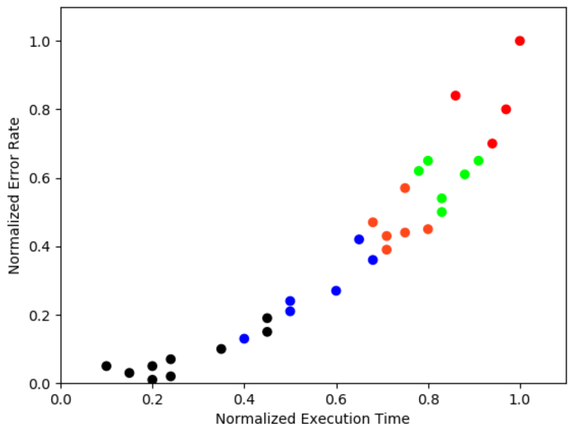

Fig. 6. Sample Normalized Time versus Error Rate Showing τ.

Applying k-NN to the τ data gives us the final graph in Figure 7. We will use this data to feed a τ value to the mediation system to limit the amount of LangSec modification done. Note that this graph is only a sample; we plan to have a much more extensive training data set and would weight and smooth the data before applying k-NN to avoid overfitting issues.

We wrote a Python-based prototype data evaluation system that handles the following:

- It accepts the execution time and error rate metrics to input into the scatter plot and feed the mesh graph for the k-NN calculations.
- It has a solver that can accept an error rate or execution degradation amount and determine the other value by accessing the k-NN data. It will also output the τ to set in the mediation system to obtain those values.

The next step is to begin building the data mediator components; we prefer to use existing code to reduce development time, but we will develop code if we cannot find suitable packages. We are investigating the following software for incorporation into the mediator:

- Fast static taint analyzers like the Taint Rabbit [16] for the mediation system. We also need a dynamic taint analysis framework like Dytan [14] that can use fuzzer data generated by the mediation system.
- Evolutionary data fuzzers that are data pattern-aware such as VUzzer [17].
- Continued evaluation of Katai Struct [18] for the SDL. It is flexible and works with many programming languages.

The graphical representation of the source code tree data and routine boundaries are the first components we are building. Our current toolchain compiles the complete target application with *clang* [19] with the *-fembed-bitcode* flag, which saves the intermediary LLVM bitcode in the application. We are also

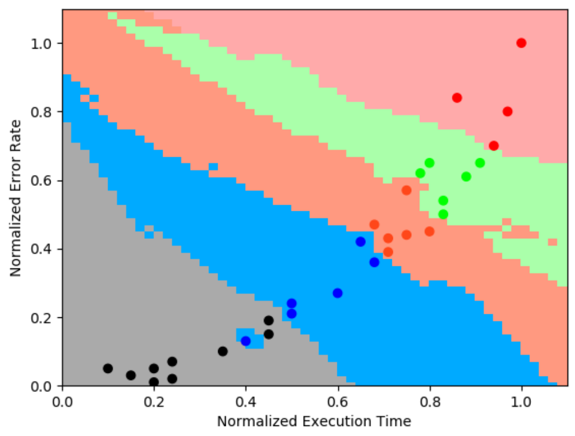

Fig. 7. Sample Normalized Time versus Error Rate with k-NN Analysis.

evaluating *whole-program-LLVM* [20] as an alternate way of generating the bitcode.

A stub program, *llvm2cpg* [21], converts the bitcode found in the executable to a code property graph (CPG) file suitable for importing into the Joern graphing system [22].

As mentioned above, our initial parser/combinator will be Hammer; even though it only outputs C/C++ code, it is a well-understood system. Another possibility is using Nom [23] if we test with Rust [24] executables. We will evaluate other parser/combinators as well.

V. Conclusions

This paper describes a novel algorithm that automatically inserts LangSec statements in a mediation flow graph based on data structures. We also use Pareto analysis to prioritize the most-used data structures to find a user-selectable balance point between security and performance. We are currently performing initial experiments to demonstrate that this approach is a viable one.

Acknowledgment

Thank you to the LangSec 2021 reviewers who gave us constructive criticism and a few new research avenues.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001119C0075 and Contract No. HR001119C0121. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA)

Appendix

The Pareto Distribution and the Pareto Principle

Vilfredo Pareto (1848 – 1923) was an Italian economist and sociologist. In his initial 1896 book, *Cours d'économie politique* [25], Pareto observed that 20% of the population owned approximately 80% of the land in Italy. From this observation, he built a large set of income data from all over Europe and North and South America. From the data, he extrapolated that the distribution of income follows a logarithmic pattern. which he described as:

$$\log(N) = \log(A) + \alpha \log(x)$$

N is the number of people who receive incomes higher than x, and A and α are constants. α is known as the Pareto index, which in the case of income shows the more significant the α, the smaller the number of high-income people. This formula is known as the *Pareto distribution* and is used for many types of empirical studies.

Pareto's work was re-discovered in the 1930s by Joseph M. Juran, a quality engineer who noticed that production defects were not equal in frequency. Juran realized that much of Pareto's work correlated to a variety of production issues. By ordering the defects by frequency, he found that only a few defects caused most of the issues.

Juran coined the term "Pareto Principle" or the "80/20 rule" in the 1950s as part of his *Quality Control Handbook* [6] as *en hommage* to Pareto's observations.

The Principle states that in a result, only the "vital few" cause the bulk of the results, while the more significant number of contributors, the "useful many," provide much less. Therefore the vital few should receive priority.

One way of finding the vital few contributors is to create a Pareto chart or Pareto histogram. The steps to create the chart are paraphrased from [6, Section 19.3.15.1.2]:

1) Reorder the contributors from the largest to the smallest.
2) Draw and label the vertical axis from 0 to the total or just beyond.
3) Draw and label the horizontal axis. List the contributors from largest to smallest, going from left to right.
4) Draw a histogram to represent the magnitude of each contributor's effect.
5) Draw a line graph to represent the cumulative total.
6) Analyze the diagram and look for a slope-of-the-line breakpoint on the cumulative graph.

An example is shown in Figure 8. Here we count up the number of times a routine is called place them in order on the graph from highest number of calls to lowest. We then draw a line (blue in the figure) showing the cumulative total. Looking at the line, we see that the curve's knee is at routine D (red line on the figure), which divides the vital few from the useful many.

Looking at this from a security-performance balance point, modification of routines A-D covers 88.7% of the total number of calls made, and if we narrowed that to just routines A-C, we still cover 73.6% of the total calls.

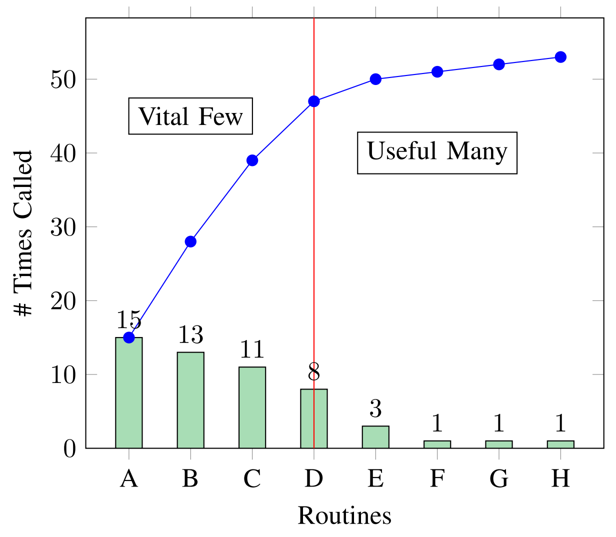

Fig. 8. Sample Pareto chart.

References

[1] Equinix, Inc., "Global Interconnection Index Volume 3," 2019. [Online]. Available: https://equinix.box.com/shared/static/tup5ig8afridlc79lngw6dom326522tz.pdf

[2] F. Momot, S. Bratus, S. M. Hallberg, and M. L. Patterson, "The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them," in *2016 IEEE Cybersecurity Development (SecDev)*, Nov. 2016, pp. 45–52.

[3] langsec.org, "LangSec: Recognition, Validation, and Compositional Correctness for Real World Security," 2011. [Online]. Available: http://langsec.org/bof-handout.pdf

[4] Voltaire, "La Bégueule," in *Romans de Voltaire suivis de ses contes en vers*. Garnier, 1922, pp. 541–548.

[5] M. Juetten, "Failed Startups: Juicero," Nov. 2018, section: Entrepreneurs. [Online]. Available: https://www.forbes.com/sites/maryjuetten/2018/11/27/failed-startups-juicero/

[6] J. A. D. Feo, *Juran's Quality Handbook: The Complete Guide to Performance Excellence, Seventh Edition*. McGraw-Hill Education, 2017. [Online]. Available: https://www-accessengineeringlibrary-com.dartmouth.idm.oclc.org/content/book/9781259643613

[7] D. Muthukumaran, S. Rueda, H. Vijayakumar, and T. Jaeger, "Cut me some security," in *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, ser. SafeConfig '10. Chicago, Illinois, USA: Association for Computing Machinery, Oct. 2010, pp. 75–78. [Online]. Available: https://doi.org/10.1145/1866898.1866911

[8] L. Pike, "Post-Hoc Separation Policy Analysis with Graph Algorithms," in *Workshop on Foundations of Computer Security (FCS'09). Affiliated with Logic in Computer Science (LICS)*, Aug. 2009, p. 15. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.439.3755&rep=rep1&type=pdf

[9] D. King, S. Jha, D. Muthukumaran, T. Jaeger, S. Jha, and S. A. Seshia, "Automating Security Mediation Placement," in *Programming Languages and Systems*, ser. Lecture Notes in Computer Science, A. D. Gordon, Ed. Berlin, Heidelberg: Springer, 2010, pp. 327–344.

[10] N. Talele, J. Teutsch, T. Jaeger, and R. F. Erbacher, "Using Security Policies to Automate Placement of Network Intrusion Prevention," in *Engineering Secure Software and Systems*, ser. Lecture Notes in Computer Science, J. Jürjens, B. Livshits, and R. Scandariato, Eds. Berlin, Heidelberg: Springer, 2013, pp. 17–32.

[11] Y. Li, R. Strom, and C. Dorai, "Placement of replicated message mediation components," in *Proceedings of the 2007 ACM/IFIP/USENIX international conference on Middleware companion*, ser. MC '07. New York, NY, USA: Association for Computing Machinery, Nov. 2007, pp. 1–2. [Online]. Available: http://doi.org/10.1145/1377943.1377946

[12] N. S. Altman, "An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, Aug. 1992, publisher: Taylor & Francis. [Online]. Available: https://amstat.tandfonline.com/doi/abs/10.1080/00031305.1992.10475879

[13] K. Underwood and M. E. Locasto, "In Search of Shotgun Parsers in Android Applications," in *2016 IEEE Security and Privacy Workshops (SPW)*, May 2016, pp. 140–155.

[14] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis*, ser. ISSTA '07. New York, NY, USA: Association for Computing Machinery, Jul. 2007, pp. 196–206. [Online]. Available: https://doi.org/10.1145/1273463.1273490

[15] M. L. Patterson, "Hammer, Parser combinators for binary formats in C," May 2019. [Online]. Available: https://github.com/UpstandingHackers/hammer

[16] J. Galea and D. Kroening, "The Taint Rabbit: Optimizing Generic Taint Analysis with Dynamic Fast Path Generation," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 622–636. [Online]. Available: https://doi.org/10.1145/3320269.3384764

[17] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *Proceedings 2017 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[18] Kaitai team, "kaitai-io/kaitai_struct," Nov. 2020, original-date: 2016-02-20T15:17:00Z. [Online]. Available: https://github.com/kaitai-io/kaitai_struct

[19] LLVM Project, "Clang C Language Family Frontend for LLVM." [Online]. Available: https://clang.llvm.org/

[20] T. Ravitch, "travitch/whole-program-llvm," Apr. 2021, original-date: 2011-07-02T15:44:44Z. [Online]. Available: https://github.com/travitch/whole-program-llvm

[21] A. Denisov, "ShiftLeftSecurity/llvm2cpg," Mar. 2021, original-date: 2021-02-19T20:46:51Z. [Online]. Available: https://github.com/ShiftLeftSecurity/llvm2cpg

[22] ShiftLeft.io, "Joern Open Source Query Engine for C/C+," 2020. [Online]. Available: https://joern.io

[23] G. Couprie, "Nom, A Byte oriented, streaming, Zero copy, Parser Combinators Library in Rust," in *2015 IEEE Security and Privacy Workshops*, May 2015, pp. 142–148.

[24] S. Klabnik and C. Nichols, "The Rust Programming Language," 2018. [Online]. Available: https://doc.rust-lang.org/book

[25] G. Bousquet and G. Busino, *Cours d'Économie Politique: Nouvelle édition par G.-H. Bousquet et G. Busino*. Librairie Droz, 1964, vol. 1.