Jahrbücher für Nationalökonomie und Statistik manuscript No.
(will be inserted by the editor)

Randomization in Online Experiments

Randomisierung im Online Experimente

Konstantin Golyaev
Konstantin.Golyaev@Microsoft.com[a]

C1, C8, C9

Big Data, data science, Internet randomized experiments, A/B testing, hash functions

Zusammenfassung

Most scientists consider randomized experiments to be the best method available to establish causality. On the Internet, during the past twenty-five years, randomized experiments have become common, often referred to as A/B testing. For practical reasons, much A/B testing does not use pseudo-random number generators to implement randomization. Instead, hash functions are used to transform the distribution of identifiers of experimental units into a uniform distribution. Using two large, industry data sets, I demonstrate that the success of hash-based quasi-randomization strategies depends greatly on the hash function used: MD5 yielded good results, while SHA512 yielded less impressive ones.

> Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.
>
> John von Neumann

1 Motivation and Goals

Many important questions in social sciences involve establishing cause-and-effect relationships—for example, whether an increase in the minimum wage increases the

[a] I thank the James M. Kilts Center for Marketing at the University of Chicago Booth School of Business for making the Dominick's data available. The views expressed in this article are those of the author and not of Microsoft Corporation. For comments and suggestions on previous drafts of this paper, I thank the co-editor, Harry J. Paarsch, as well as two anonymous referees. All remaining errors are, of course, mine.

unemployment rate among low-skilled workers. Establishing causal links makes reliable predictions concerning the potential effects of policy changes possible. Determining causal relationships using only observational data can, however, be tricky because of the omitted variables problem: one has to establish beyond reasonable doubt that only a change in the cause could have resulted in the observed effect. The great English biologist and statistician Sir Ronald A. Fisher forcefully made the case against using observational evidence to establish causal relationships in (5).

Randomized experiments provide a way to control for potential confounding factors. Rather than changing the cause values for all experimental units and contrasting the values of their effects before and after the change, one randomly divides all units into two groups, varying the cause for only one of the groups. Since, thanks to the random assignment, the differences between the groups are not systematic, they will average out; any changes to the effect are driven by the changes to the cause, thus establishing the causal link. Among technology companies that conduct their business over the Internet, randomized experiments have proliferated recently under the name A/B testing.

Today, the main way to implement randomization on computers is to use pseudo-random number generators. Even though such pseudo-random numbers are both easily reproducible and mostly indistinguishable from true random ones, several reasons exist (discussed below) that make using them impractical for Internet-based A/B testing. Instead, hash functions are used to transform the distribution of experimental unit identifiers into something that resembles a uniform distribution, which is then used to assign units into treatment and control groups according to the experiment's design.

This paper is motivated by the observation that the choice of hash function matters. Hundreds of different hash functions exist, but some are more appropriate for quasi-randomization applications than others. Some hash functions, when applied to only a subset of all possible inputs, can yield a distribution that is nowhere close to the uniform distribution. I illustrate this fact by applying two popular hash functions to two data sets having large numbers of products. The MD5 hash function yielded distributions that are virtually indistinguishable from the uniform's, while on the same data sets, the SHA512 hash function produced considerably less-than uniform distributions as measured by formal statistical tests.

The remainder of this paper is organized as follows: In Section 2, I provide an overview of what randomness means and how it is used for experimentation. Next, in Section 3, I discuss how experiments can be conducted over the Internet. After that, in Section 4, I outline how hash values are used *in lieu* of pseudo-random numbers when assigning treatment status in experiments. Finally, in Section 5, I present the results from applying the hash-based quasi-randomization technique to two large data sets, while I conclude in Section 6.

2 Overview

Randomness is surprisingly difficult to define clearly. In the same spirit of defining darkness as the absence of light, the Merriam-Webster dictionary defines randomness as the absence of any predictable pattern. The following story may be helpful: Imagine a person sitting on the lake shore, throwing stones into the lake without any purpose. If every subsequent stone is equally likely to hit the water anywhere, then one might claim that the stones are thrown randomly. Of course, many different factors can affect the final distribution of stones' splashes, such as thrower strength, heterogeneity in stones, wind speed, thrower stamina, and so forth. In short, it is difficult to construct a clean example of a perfectly random process.

Perfect randomness is largely a theoretical abstraction invented by statisticians to simplify representing complex realities using simple models. Even though individual random events are unpredictable, the relative frequencies of different outcomes averaged over large numbers of events can be predicted. Thus, another way to view randomness is as a measure of the risk of outcomes, rather than complete unpredictability.[1] Of the many fields in applied statistics that rely on the notion of randomness, few are as critically dependent on it as the field of experimental design and evaluation. Since the publication of Fisher's pathbreaking book (5), statisticians have recognized that randomization is the key to proper causal inference. Questions such as "Does x cause y?" are critical to understanding how the world works, but can be fiendishly difficult to answer. At the heart of the challenge is the problem of omitted variables bias. When one observes a change in y and seeks to determine whether a change in x caused it, excluding the possibility of a change in some other factor z that one never observes is nearly impossible. In reality, however, z could have been the true cause of the change in both x and y. In fact, Fisher used this rhetoric to argue that smoking need not cause lung cancer: other confounding factors could be the source of both smoking and cancer. Today, Fisher's error is well understood, but if an intellectual giant of Fisher's stature can get a little confused, then the rest of us need to be very careful when attempting to establish causality.

Randomization is considered the best method available to establish causality, e.g. (4). Suppose one seeks to quantify the causal impact of a change in x on y. The best way to accomplish this is as follows:

1. Put together a sample of I units for which both x_i and y_i can be reliably measured.
2. Randomly split the sample into two halves, labeling them "treatment" and "control."
3. Change the values of x in the treatment group, leaving the values of x in control group unchanged.
4. Wait for some reasonable amount of time, and then measure y in both treatment and control groups.
5. Compare the average of y values between treatment and control groups to form conclusions.

[1] I am making a distinction between risk and uncertainty, using the term risk in the sense proposed by Frank H. Knight in his classic book (6). In other words, Knightian risk can be characterized by a probability distribution, whereas Knightian uncertainty cannot.

Central to implementing the above plan is randomization. Even though flipping coins or rolling dice can be entertaining, such methods of randomization have two major drawbacks. First, they are time-consuming, particularly for large I. Second, they are fundamentally non-reproducible, which is an undesirable property for a scientific process. Yet, herein lies a quandary: reproducibility is inconsistent with true randomness.

In the past, people relied on tables of random numbers that were carefully constructed to ensure randomness, yet offered the ability to reproduce the sequence of random draws when needed. Today, one relies on the modern counterparts of such tables—pseudo-random number generators, sophisticated computer programs that generate *deterministic* sequences of numbers that are mostly indistinguishable from true random sequences. By fixing a constant, commonly referred to as the *seed*, it is possible to reproduce the sequence; by supplying different seeds, one can obtain completely different sequences. Section 7.10.3 of (9) provides an overview of pseudo-random numbers and the software used to produce them. Without such software, it is impossible to perform experiments at scale.

3 Online Experimentation

3.1 Terminology

In this section, I briefly outline different types of experiments that are commonly performed by technology companies on the Internet. Before discussing the details, however, I have found it useful to define several key terms that will be used throughout the paper. Let me first, however, acknowledge that my presentation has been heavily influenced by the research of several of my colleagues at Microsoft, some of which is documented in (7).

First, an experiment is defined by its treatment logic. In the majority of applications, the existing state is typically chosen as the baseline (control), while a proposed change is implemented as the treatment. A bewildering variety of potential treatments exists, the simplest being a small tweak in webpage layout. Highlighting different attributes of an offer, such as price or feedback from customers who previously purchased it, are other examples. Less obvious examples involve changes to the back-end algorithms that determine what users see on a page, for instance, result ranking algorithms for seach engines, like Bing or Google. Describing different kinds of treatments is its own paper.

For the remainder of this paper, I shall focus on experiments with a single treatment and asssume that each unit (see below) is *ex ante* equally likely to fall into either the treatment or the control group. Among technology companies, experiments like these are commonly referred to as A/B tests. Although the name suggests that only two variants (A and B) are being contrasted for the purpose of finding which one works better, it is now common to refer to experiments with several treatments as A/B tests, too. In addition to A/B tests, A/A tests exist as well. As that name suggests, in an A/A test, two identical versions are pitted against each other; the main purpose of A/A tests is to assess the quality of experimentation software infrastructure. By

design, no differences between treatment and control should be found in an A/A test; if a significant difference is found, then its most likely cause is defective software.

Second, every experiment must have a success metric (or criterion). Without one, it is impossible to decide whether an experiment delivered useful results. Put simply, when no destination is set, one cannot determine when the journey has ended. Typical success metrics tend to be closely related to business objectives; examples include net revenue or the conversion rate (the fraction of webpage visitors who have placed orders) as well as more technical metrics, such as latency (how long it takes for a webpage to render). The success metric must be measurable: however useful it may be to improve customer sastisfaction, that sentiment is notoriously difficult to measure. Moreover, experience shows that a successful experiment involves a single success metric. In a business environment, such a requirement can be difficult to effect in practice because different stakeholders can often have conflicting objectives. In addition, for reasons that have more to do with corporate politics rather than science, which are beyond the scope of this paper as well, it is best to reconcile such conflicting objectives *before* the experiment rather than *after*.

Third, the unit of experimentation must be chosen. The success metric must then be computed for every unit in the experiment, so one can compare how the treatment and control groups differ in their experiences. Most commonly, every website visitor is a single unit of experimentation; for example, in an online store, Alice and Bob see different page layouts for the same item or receive different ranked search results for the same input query. Alternatively, an online store may set a product as the unit of experiment, and then examine competing pricing strategies for treatment and control.

Finally, for the purpose of making a decision concerning the final outcome of the experiment, choosing a confidence level is important. Given the treatment definition, the success metric, and data concerning units of experimentation, the statistical underpinnings required to make a decision are usually relatively straightforward. The significance level enables making a decision concerning how large a difference between the treatment and the control groups is deemed too large to be driven purely by sampling variation.

An important, but frequently overlooked, aspect of online experimentation is the relative ease of rolling back the changes. For instance, consider an experiment in which the treatment appears to improve the success metric dramatically on average, but results vary considerably across individual experimental units. One could argue that it may be worthwhile to launch the treatment version as the new default and monitor the results closely, being ready to roll back to control if improvements do not occur. This strategy may be too risky in an experimental setting where poorly chosen treatments may have long-lasting adverse impact on subjects, such as in tests of drug by pharmaceutical firms.

3.2 Types of Online Experimentation

Classifying online experiments into two major groups based on the unit of experimentation makes analysis clearer. Most Internet experiments focus on website visitors as

the experimental units—that is, user-based experimentation. Item-based experiments are less common, even among firms like Amazon.com, which can perform experiments across its vast product catalog, or Netflix, which can lever its expansive library of movie titles to the same end.

What are the advantages and disadvantages of the user-based versus the item-based approaches? In general, user-based experiments are more attractive because the sample sizes in such experiments naturally grow as new visitors arrive at the website. Every participating user receives a consistent user experience on the website, but different users' experiences will vary. By contrast, item-based experimentation admits a uniform experience for all users interacting with a product, but the experiences across products can differ.

Pricing experiments are good examples of when the item-based approach is superior to user-based approach. Randomly perturbing the price of a product charged by an online store is an excellent way to eliminate the endogeneity that plagues researchers seeking to estimate demand elasticities. Unfortunately, users tend to resent such experiments: that some pay more than others for exactly the same product is perceived as unfair, which can cost the firm money. (Remember, the goal of experimentation is to improve firm performance, not to alienate the customer base.) Within an item-based framework, prices for an entire group of products are randomly perturbed; every user faces these perturbed prices, thus eliminating the unfair pricing.

In addition to segmenting experiments according to experimentation unit, it is also helpful to consider experimentation layers—that is, to distinguish between front-end and back-end experiments. A front-end experiment usually involves a change in the layout of a webpage, which can be as comprehensive as a complete website overhaul or as minimal as a change in the text on the button that is displayed to the visitor—for instance, "Donate Now" versus "Learn More". By contrast, a back-end experiment is usually concerned with tweaking the algorithms resposible for generating webpage contents, such as search engine results in response to a particular query.

The combinations of experimentation unit and layer are presented in Table 1.

Tabelle 1 Examples of Different Types of Online Experiments

		Experimentation Layer	
		Front End	Back End
Experimentation Unit	User	Change page text or design	Alter search algorithm
	Item	Highlight certain item attributes	Vary prices for some products

3.3 Software Implementation

On the surface, nothing prevents the researchers responsible for online experiments from using pseudo-random number generators to determine treatment assignments. State-of-the-art methods for generating sequences of pseudo-random numbers exist and are readily available. For example, the Mersenne Twister algorithm, developed by (8), is used to generate pseudo-random numbers in `Python` and `R`; implementations of this algorithm exist for `C/C++` and `Java` as well.

The traditional experimentation paradigm, however, involves a number of assumptions that greatly facilitate random treatment assignments. First, the experimentation population is usually of fixed size; that is, the treatment assignment for all needs to happen just once all at the beginning of experiment. Second, once treatments have been assigned, it is usually straightforward to identify the treatment status of every subject.

By contrast, on large websites, many experiments may be taking place simultaneously; a single customer can be enrolled in several tests. Moreover, it is often desirable to be able to adjust the percentage of population that falls into the treatment group.

In an online setting, these factors do not play well with the way that pseudo-random numbers are generated. In user-based experiments, the population under experimentation is usually defined as all visitors to a particular set of webpages over the course of the experiment. Almost by definition, the size of the group is unknown *a priori*. As such, one-shot assignment is infeasible.

This problem is aggravated by the fact that most large websites rely on fleets of servers to handle incoming traffic from the Internet. Generating pseudo-random numbers in a distributed-computing environment is difficult: a naïve strategy for selecting pseudo-random number generator seeds can cause the outputs to be correlated. An extreme example would involve setting the seed to the same value on every server, thus obtaining identical sequences of pseudo-random numbers.

Identifying and keeping track of the treatment status of every participant in the experiment is also challenging. On some websites, like Facebook, it is easy to identify returning users because they sign in. Most websites, however, do not require visitors to authenticate their identity when browsing, which makes identifying returning visitors difficult.

One way to detect returning visitors on the Internet involves using cookies, which are small text files created by the website that are stored on the user's device. Some users do not accept cookies; others delete them periodically. In either of these circumstances, return identification is made virtually impossible. Moreover, a single visitor can use several different devices to access a website, but cookies are inherently tied to a device. Even when users accept cookies and are kind enough to use only a single device to access the website, randomization based on pseudo-random numbers is more difficult than it may seem. Here's why.

Initially, imagine that the website is run on a single server. In principle, one could call a pseudo-random number generator every time a new user arrives, and assign a treatment status to the user based on that pseudo-random draw. One would then need to record the treatment status for each user and store it in some lookup table, where it can be easily accessed later. When a new visitor arrives, one must check whether this user has already been assigned to a treatment using the lookup table, and generate a status assignment if one has not been previously made. If website traffic is managed by more than one server, then this lookup problem quickly becomes intractable: as more and more users arrive, a balance must be struck between inserting new records into this table for users who have not been assigned to a treatment group and ensuring that the table is up-to-date and available for every machine in the website fleet, so every user who already had been assigned is handled accordingly. The CAP theorem

from the database theory guarantees that this will eventually become impossible to implement; see (1).

Admitting several experiments per user exacerbates the problems: in an online setting, tens of concurrent experiments can exist. For example, Amazon.com could be testing changes to layouts of product detail pages in the shoes category, while simultaneously experimenting with alternative recommendation models for video streaming. Any customer shopping for shoes will be automatically enrolled in the first test; if the same person decides to stream a movie, then that person is likely to fall into the recommendation algorithm experiment as well. When treatment assignments for every experiment are implemented using pseudo-random numbers, it becomes necessary to generate and to store users' treatment statuses for every test in which they are enrolled. Reusing the random draws will almost surely introduce unwanted correlations into treatment assignment, and undermine inference when assessing the statistical significance of experimental results.

Scaling up and down treatment percentages is a desirable feature to have for back-end experiments—for instance, when old software is reimplemented using a new framework which, in theory, should produce identical results. Rather than launching the new code for all users at once, it is typically initially enabled for a small random fraction of incoming traffic, on whom its performance is carefully monitored, to avoid massive performance degradation. If the new code performs well on a small subset of traffic, then it is dialed up incrementally to a larger percentage and is continued to be monitored for potential issues. Being able to dial up treatment percentages randomly helps to ensure that the problems are not arising simply because of some systematic differences in the traffic received by the new system.

For all of the above reasons, and more, rather than relying on pseudo-random numbers, computer scientists have instead developed quasi-randomizing techniques that are based on the hashing algorithms described in the next section.

4 Hashing

The main idea behind using hash functions for quasi-randomization is the following: map the data into a uniform distribution using a deterministic function. Formally, a hash function is any function that can map inputs of arbitrary size to outputs of fixed size. The outputs returned by such function are called hash values, but they are also frequently referred to as hash codes, or just hashes, for short. In Appendix A, I provide a simple, yet detailed, example borrowed from (2). One way to use hash functions in computer science is to construct hash tables, which are data structures optimized for rapid retrieval of stored information.

In principle, it is straightforward to a use hash function as a quasi-randomizing device. Consider the case of a user-based experiment. First, a unique string is used to identify the experiment must be chosen. Second, for every user who is selected to be a part of the experiment, their unique user identifier is combined with the experiment identifier, for instance, "`User1_Experiment2`". Third, a hash function is applied to this string. Finally, for the equal treatment-control split case, one could assign the odd hashes

to the treatment group and the even hashes to the control, or vice versa. In general, once the hash value has been produced, more sophisticated allocation strategies can be easily employed. For example, if the entire set of experimental units is known in advance, as it is frequently the case in an item-based experiment, then one can identify the largest hash value and rescale all hashes to fall into the unit interval, thus reducing the assignment problem to the case where it is possible to make uniform pseudo-random draws for each unit.

Of course, uniformity and randomness are not necessarily interchangeable. In fact, proper pseudo-random generated sequences of numbers frequently appear to be non-random to an untrained eye. Usually this happens because people tend to underestimate probabilities of runs. Consider the following example. Suppose one uses a pseudo-random number generator to randomly assign test subjects to either treatment or control with equal probability. Examining the data reveals that this process has assigned treatment to seven subjects in a row. One might argue that the probability of seven treatments in a row is 0.5^7 or about 0.008. But this line of reasoning grossly underestimates the probability of a run of 7 identical assignments. If someone asked the probability that the next 7 assignments would all be treatments, then 0.5^7 would be the right answer. But that is not the same as asking whether an experiment is likely to see a run of length 7 because the run could start any time, not just on the next assignment. More details about likelihoods of runs can be found in (11). In contrast, consider a process that can only take values 0 or 1. A deterministic rule $y_t = 1 - y_{t-1}$ will generate extremely uniform values with enough draws, even though there is nothing random about any of them.

Hash-driven treatment assignment strategies have some major advantages: First, hashing turns treatment assignment and lookup into a "stateless" problem. Knowing the experiment identifier and the unit identifier is sufficient to compute the hash value for the unit, together with the treatment assignment function this completely alleviates the need to remember the treatment status for every unit: because hash functions are deterministic, applying them to the same inputs is guaranteed to produce the same outputs. Second, this strategy trivially scales out to multiple experiments per user: all it takes is to come up with a different experiment identifier for each new test. This ensures that the hash value of the combination of unit identifier and experiment identifier will vary. Third, for user-based experiments, this strategy easily accommodates new units coming into the experiment. Provided every subsequent user has a well-defined user identifier, that user can be combined with the experiment identifier and hashed, and the treatment status can be assigned.

Of course, disadvantages exist when using hash functions *in lieu* of pseudo-random number generators, too. Most of these disadvantages derive from the same root cause: some hash functions work better than others. Hundreds of hash function exist; their properties vary, depending on the application. Two facts warrant special attention—local sensitivity and the possibility of collisions.

To discuss local sensitivity, it is often helpful to imagine that the hash function is a continuous (in the mathematical sense of the term) function of its argument; that is, small perturbations in inputs do not result in large abrupt swings in the values of outputs. Although there is nothing sinister about continuity *per se*, consider what can happen if unit identifiers all adhere to some common structure. In this case, hash

values for units with similar identifiers can end up being similar as well; a non-trivial correlation can be induced between treatment and control units.

The possibility of hash-value collisions introduces similar challenges. A collision in hash values occurs when two different inputs are mapped to exactly the same output. The example in Appendix A is designed to produce collisions. A well-chosen hash function minimizes collisions. A poor choice of hash function can yield a distribution of hash values that has mass points, which can in turn induce correlation between treatment and control units.

In my experience, the above problems are more likely to happen in item-based as opposed to user-based experiments because in user-based experiments identifiers are frequently generated on the fly when the user lands on the first webpage. These identifiers are typically long strings of digits that are determined in part by the system clock of a server. To the extent that arrivals are random, so too are the hash values. In addition, as more users enroll in the experiment, an increasing array of possible user identifiers is likely. By contrast, item identifiers are frequently determined according to a particular logic, and the experimental populations are selected in advance. As such, some patterns in item identifiers may be present in the experiment; with a poorly chosen hash function, treatment assignments may fail to be as good as random ones, pseudo-random ones that is.

To illustrate the potential problems with hash-based treatment assignments in an item-based framework, I investigated applying this quasi-randomizing strategy to two data sets having large numbers of products. Two popular hash functions were used: MD5, proposed in (10), and SHA512, proposed in (3). Both of these functions are complex enough to omit the details on how they are implemented. For my purposes, however, I only needed to compute them for an arbitrary input.

5 Application

In this section, I present results derived from using two different hash functions on an item-based experimentation setting involving two data sets. First, I consider the Dominick's data set, which has been popular among marketing researchers. I then repeat the analysis using products from a large monthly panel data set of laptop sales in North America. This data set was purchased by Microsoft for market analysis and was made available internally for general research purposes. For both data sets, I conducted the analysis according to the following steps:

1. I extracted the complete set of unique product identifiers from the data. In case of Dominick's data, I collected all of the 18,003 unique universal product codes (UPCs) that were made publicly available to researchers.
2. I appended the same string "ExperimentName" to each product identifier and applied a hash function to each of the resulting strings. This resulted in a 32-character string that contained the hexademical representation of hash value. A hexademical number can take sixteen possible values, from 0 to 9, as well as letters A, B, C, D, E, or F. For example, UPC "001192603016", which corresponds to

the product "Caffedrine Caplets 1", became "74288c271f3f75163234e0fb8cc4d8fa" when passed through the MD5 hash function.
3. Since every hash value is simply a huge number, I rescaled all of them by dividing each by the largest value possible on the computer, which ensured that the rescaled hash values lived in the unit interval and preserved their distribution.
4. Under the null hypothesis that hashing can be used *in lieu* of randomization, the rescaled distribution should resemble closely the uniform distribution. I formally tested this hypothesis, first using the Kolmogorov–Smirnov (KS) test, and then by discretizing both distributions into 20 bins with the step size of 0.05 and applying Pearson's χ^2 test.

My entire analysis was implemented in the `R` programming language. Reading and cleaning the product identifier data were relatively straightforward exercises in data munging. Similarly, implementing the KS and χ^2 tests were undemanding tasks. The only bottleneck involved the scaled distribution of hash values of the products: the `digest` package in `R` provides access to a number of popular hashing algorithms, including MD5 and SHA512. Unfortunately, most hash functions return numbers that can results in numerical overflow errors for standard `R` data types, including the ubiquitous double-precision floating point type, often referred to as double. A package, having the cryptic name `Rmpfr`, which stands for "R Multiple Precision Floating-Point Reliable", was used to handle such large numbers correctly, without losing precision. Once rescaled, the hash values could then be represented by `R`'s double data type without incident.

5.1 Data Descriptions and Summaries

I used the UPC data provided by Dominick's Finer Foods to James M. Kilts Marketing Center at the University of Chicago Booth School of Business. From 1989 to 1994, the Graduate School of Business at the University of Chicago and Dominick's Finer Foods entered into a partnership for store-level research into shelf management and product pricing. Randomized experiments were conducted in more than 25 different categories throughout all stores in this 100-store chain. One byproduct of this research cooperation was a data set that concerned approximately nine years of store-level data involving the sales of more than $3,500$ UPCs. These data are unique in the breadth of coverage and for the information available on retail margins. None of the products in the UPC files is available for sale any longer.

In my analysis, I only focused on the UPC files. The data set contains $18,003$ unique UPCs that span 28 categories—such as beer, cereals, cheeses, and so forth. Although I expected UPCs to be twelve digits long, the majority of them only had ten digits; some had as few as three digits. To unify records, I prepended the appropriate number of leading zeros to each UPC to ensure each is exactly twelve digits long.

I also analyzed the data concerning laptop sales in North America during the period of July 2012 through April 2016. This data set contains $51,319$ unique products that have several attributes. Of these, 53 percent are laptops, 40 percent are desktops, 3.4 percent are tablets, with the remainder being smartphones. These products were manufactured by 122 different companies, including Dell and Lenovo. 84.5 percent of

these products came with some form of Windows operating system, and about 7.1 percent were OS X machines. For this analysis, I treated all products in this data set equally; excluding some or all of the non-PC products had no meaningful impact to results.

5.2 Results

I present the results from applying the KS and Pearson's χ^2 tests in Table 2. To illustrate that the choice of hashing algorithm matters, I investigated two popular hash functions: MD5 and SHA512. Results from applying the MD5 algorithm do not reject the notion that the distribution of scaled hash values is statistically indistiguishable from the uniform distribution for both data sets. In contrast, when I used the SHA512 hashing algorithm, I obtained considerably larger values of the test statistics using the same data. The SHA512 results alone calls into question the notion of a uniform distribution of scaled hash values.

Tabelle 2 Results from Comparing Distribution of Scaled Hash Values to Uniform Distribution

		MD5 Hash		SHA512 Hash	
Data set	Test	Test Statistic	Test P-Value	Test Statistic	Test P-Value
Dominick's	KS Test	.0044	.884	.0067	.396
	χ^2 Test	14.3249	.814	31.2043	.052
PC Sales	KS Test	.0026	.867	.0056	.078
	χ^2 Test	21.2089	.385	28.4202	.099

These results suggest that the choice of hash function is important. One hash function yielded a distribution of values that is much closer to the uniform distribution than did the other. It would be desirable to repeat the above analysis with a large collection of diverse data sets to strengthen the conclusions. In the absence of such data, I approximate the above experiment by drawing 2000 bootstrap samples from each of the two data sets and repeating the analysis on each sample. In Figures1 and 2, I plot the histograms of p-values for each test and hash function combination over the bootstrap samples. Each bin has a fixed width of 0.05. I added vertical lines to point out the original p-values for the corresponding data set, test, and hash functions.

The results from the plots reinforce my conclusion, both quantitatively and qualitatively. In all four plots, a larger fraction of the SHA512 distribution mass is located in the leftmost bin, as contrasted with the MD5 distribution. In Table 3, I summarize what fraction of bootstrap samples resulted in rejection of the null hypothesis. In all cases, I am more likely to reject the null for SHA512 hashes than for MD5 hashes.

A final observation is also in order: even for MD5 hashes, the null hypothesis gets rejected on a large fraction of samples, potentially as high as 77.8 percent of them. I conjecture that these results are at least partially driven by the bootstrap approach: since all samples are drawn with replacement, it is possible that samples where the null gets rejected contain fewer *distinct* products in them, thus increasing the probability of hashing collisions.

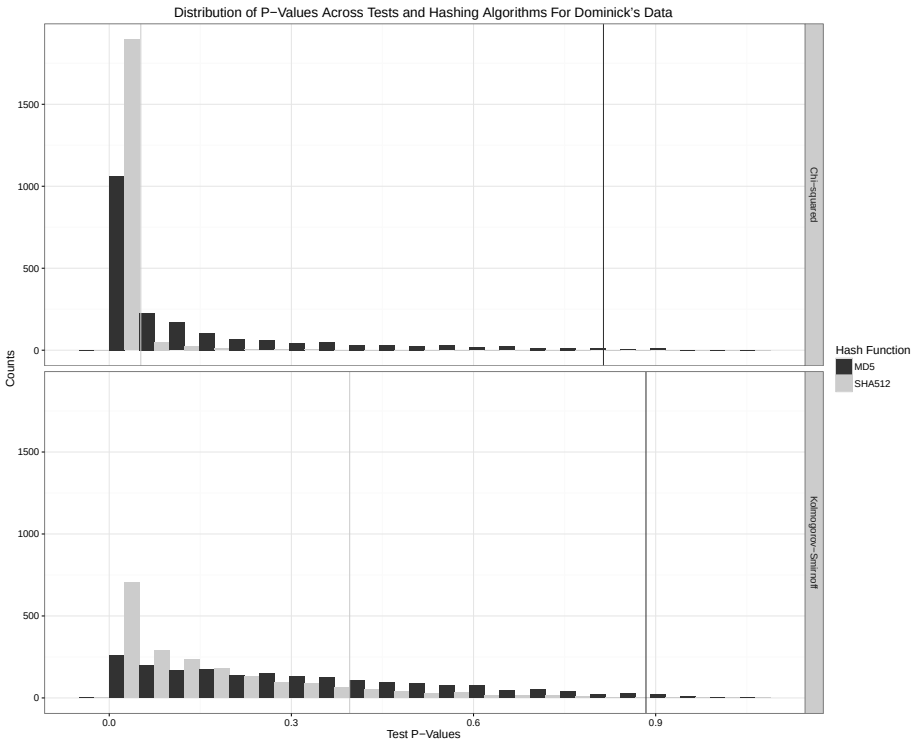

Abbildung 1 Distribution of P-Values for Dominick's Data, Based on 2000 Bootstrap Samples

Tabelle 3 Results from Bootstrap Samples of Both Data Sets

		Fraction of P-Values <= 0.05	
Data set	Test	MD5 Hash	SHA512 Hash
Dominick's	KS Test	.1305	.3555
	χ^2 Test	.5335	.948
PC Sales	KS Test	.1595	.6775
	χ^2 Test	.778	.917

Beyond these examples, in my experience, I have seen the distribution of hash values to contain mass points, as well as wide intervals with zero mass. To be clear, such deviations are not caused by the hash function alone. Rather, they obtain because of interactions between the hash function and the inputs, which may come from a limited subset of all possible inputs.

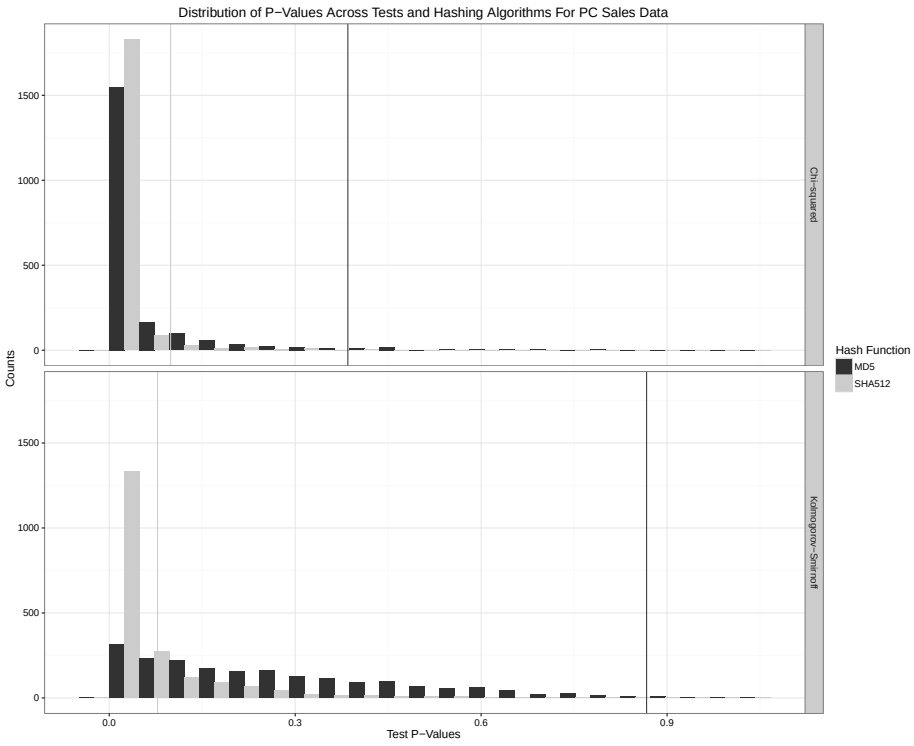

Abbildung 2 Distribution of P-Values for PC Sales Data, Based on 2000 Bootstrap Samples

6 Conclusion

Randomized experiments are considered the best method available to establish causality. Pseudo-random number generators are the best known way of generating reproducible numbers at scale that are as good as random for most practical purposes. In online settings, however, it is common to use hash functions instead of generating pseudo-random numbers. Even though hash-based quasi-randomization strategies are often viable, their performance hinges critically on the choice of hash function. Using the Dominick's data set and a large panel data set of PC sales, I demonstrated that popular MD5 hash function generated numbers that are statistically indistinguishable from uniform pseudo-random numbers. On the other hand, the SHA512 hash function generated numbers that are considerably less than uniformly distributed.

Literatur

[1] Gilbert, Seth L. and Nancy A. Lynch (2002) "Brewer's Conjecture And the Feasibility of Consistent, Available, Partition-Tolerant Web Services." ACM SIGACT News 33, 51–59.
[2] Graham, Ronald L., Donald E. Knuth, and Oren Patashnik (1994) "Concrete Mathematics: A Foundation for Computer Science." Addison-Wesley, Reading, MA, USA.

[3] Gueron, Shay Simon Johnson, and Jesse Walker (2011) "SHA-512/256." Proceedings of the 2011 Eighth International Conference on Information Technology: New Generations, pp. 354–358
[4] Harris RP, Helfand M, Woolf SH, Lohr KN, Mulrow CD, Teutsch SM, Atkins D; Methods Work Group, Third US Preventive Services Task Force (2001). "Current Methods of the US Preventive Services Task Force: A Review of the Process." American Journal of Preventive Medicine, 20 (3 Suppl)
[5] Fisher, Ronald A. (1935) "The Design of Experiments." Oliver and Boyd, Edinburgh, UK.
[6] Knight, Frank H. (1921) "Risk, Uncertainty, and Profit." Hart, Schaffner and Marx, Boston, MA.
[7] Kohavi, Ron, Roger Longbotham, Dan Sommerfield, and Randal M. Henne (2009) "Controlled Experiments On the Web: Survey and Practical Guide."
[8] Matsumoto, Makoto and Takuji Nishimura (1998) "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator." ACM Transactions on Modeling and Computer Simulation 8, 3–30.
[9] Paarsch, Harry J. and Konstantin Golyaev (2016) "A Gentle Introduction to Effective Computing in Quantitative Research: What Every Research Assistant Should Know." MIT Press, Cambridge, USA.
[10] Rivest, Ronald (1992) "The MD5 Message-Digest Algorithm." RFC 1321, RFC Editor, USA.
[11] Schilling, Mark F. (2012) "The Surprising Predictability of Long Runs." Mathematics Magazine 85, number 2, pages 141149.

Appendix

A Example of Hashing

A wonderful illustration of hashing is provided in (2). Consider a data structure that consists of collection of key-value pairs. For example, a retailer might have a database of customers and store a number of data values for each customer. Imagine a key-value store (dictionary) where keys are names of customers and values are arbitrary arrays of customer-specific data. In practice, unique, randomly-generated customer identifiers would likely serve as keys; in a large database, the likelihood of naming collisions among customers becomes too high. Using customer names as keys, however, facilitates exposition.

As an example, consider locating a particular customer within the database, specifically assume that a total of I customers are recorded in the customer database. A naïve implementation of storage and retrieval would involve maintaining a lookup table $K(i)$ for each customer $i \in 1, \ldots, I$. Searching for data concerning specific customer i^* would then involve the following steps:

1. Set $i = 1$.
2. If $i > I$, stop and return "failure".
3. If $i = i^*$, stop and return "success".
4. Increase i by 1 and go back to step 1.

In the worst-case scenario, finding data concerning customer i involves $(I + 1)$ reads from the list of keys if one decides to traverse it in this fashion. In a real-world

application, where I can easily be in billions, this method is too slow. Hashing was designed to speed up the process by storing the keys in J smaller lists.

Formally, a *hash function* maps a key $k \equiv K(i)$ into a list of numbers $h(k)$ between 1 and J. In addition, two auxiliary tables are created, $F(j)$ and $N(i)$, where $F(j)$ points to the first record in list $j \in 1, \ldots, J$, and $N(i)$ points to the "next" record after record i in its list.

As an illustration, let $J = 4$ based on the first letter of customer's name as follows:

1. $j = 1$ if first letter is between A and F,
2. $j = 2$ if first letter is between G and L,
3. $j = 3$ if first letter is between M and R, and
4. $j = 4$ if first letter is between S and Z.

Before any data are recorded, set $I = 0$. To formalize the notion of an empty list, set $F(1) = F(2) = F(3) = F(4) = -1$ In addition, set $N(i) = 0$ to denote when i is the last entry in its list. Armed with these definitions, one can begin inserting data into this new structure.

Assume that the first customer who needs to be recorded is named Nora. The hash function defined above will insert Nora's record into list $j = 3$, since the first letter of her name, N, is between M and R. Now $I = 1$, $F(3) = 1$, and all other values of F and H are so far unchanged. Let the name of the second customer be Glenn. Inserting him into the database would change I to 2 and set $F(2) = 2$, with no other changes. Now suppose that the third customer is named James. Adding him would result in $I = 3$, and $N(2) = 3$. With three records in the database the entire structure now looks as follows:

- Number of records: $I = 3$, and number of hash buckets: $J = 4$.
- Available keys: $K(1) =$ Nora, $K(2) =$ Glenn, $K(3) =$ James.
- Indices of first records in each hash bucket: $F(1) = -1$, $F(2) = 2$, $F(3) = 1$, $F(4) = -1$.
- Indices of next records in each hash bucket after the first one: $N(1) = 0$, $N(2) = 3$, $N(3) = 0$.

Fast-forwarding the example, assuming that 18 customer records were inserted into the database, things now looks like the following:

List 1	List 2	List 3	List 4
Dianne	Glenn	Nora	Scott
Ariel	James	Michael	Tina
Brian	Jennifer	Nicholas	
Francis	Joan	Ray	
Douglas	Jeremy	Paula	
	Jean		

From this example, one can see that, in the worst-case, it would take six steps to locate the record using these four lists. With 18 total records this is a three-fold speed up in search and retrieval. In the average case, the time it takes to locate a record falls from $(I/2)$ to $(1/J)$; when I and J are large, this makes a big difference. A precise search algorithm for an entry i^* would look as follows:

1. Set $j = h(i^*)$ and $i = F(j)$.
2. If $i \leq 0$, stop and return "failure".
3. If $K(i) = i^*$, stop and return "success".
4. Set $j = i$, set $i = N(j)$, and return to step 2.

To locate Jennifer in the above example with 18 records, set $j = 2$, since J is between G and L, and $i = 2$, since Glenn is the first entry in the second list. Because Glenn is not Jennifer, update i to 3, since $N(2) = 3$, that is, James is the next record in the second list after Glenn. Now, James is also not Jennifer, so i is updated to $N(3)$. The exact value would be determined by when Jennifer was added to the database, but the important part is that now $K(N(3)) = i^*$, that is, the search terminates successfully after three iterations.