

Anonymizer Functions

This paper describes a series of functions, written in the C-Programming Language that can anonymize information. Additional analysis of the functions, including risks, is detailed. These functions are presented to validate the algorithms. In turn, they are constructed similar to methods applied to objects. Note: the programs discussed are designed to operate on ASCII characters that are 8-bits in length. Different routines would be necessary to accommodate double-byte characters such as Unicode.

Objectives

The objective is to anonymize sensitive information from any number of source databases to create instances that can be used for research and other purposes. Additionally, the anonymized instances should not contain information that compromises privacy information or could be used in litigation. Yet, the translated information should allow meaningful research and to reference medical science without unnecessarily exposing sources. Ideally, the goal would be to allow researchers to label specific instances of medical events without compromising the source. This assumes the event is not already known publicly.

Forward

Medical information includes valuable knowledge that can be mined thereby advancing medicine. However, some of the information could be damaging to the parties discussed. Moreover, bad actors could use the information and through social engineering, cause damage to unwitting stakeholders. Consequently, rather than try to anticipate what needs to be anonymized, the scope of this work is to provide the tools that can anonymize typically exploitable information.

The goal is to take potentially private information and convert it such it cannot be tied back to the original person (or party). This suggests the use of a one-way-function, such as a hashing algorithm. A hashing algorithm takes data of any length and converts it into a fixed size checksum value referred to as a message digest. However, if we know what the hashing algorithms is we can apply plain text attacks to effectively recreate the source data. For instance, one attack used successfully against passwords is to take a dictionary and apply the one-way hash (or encryption) algorithm against every word. Then looking at the database of hashed passwords, any value that is found, corresponds to the dictionary word initially converted. However, if the algorithm used is not known, there is no basis for conducting a dictionary (or plain text) attack.

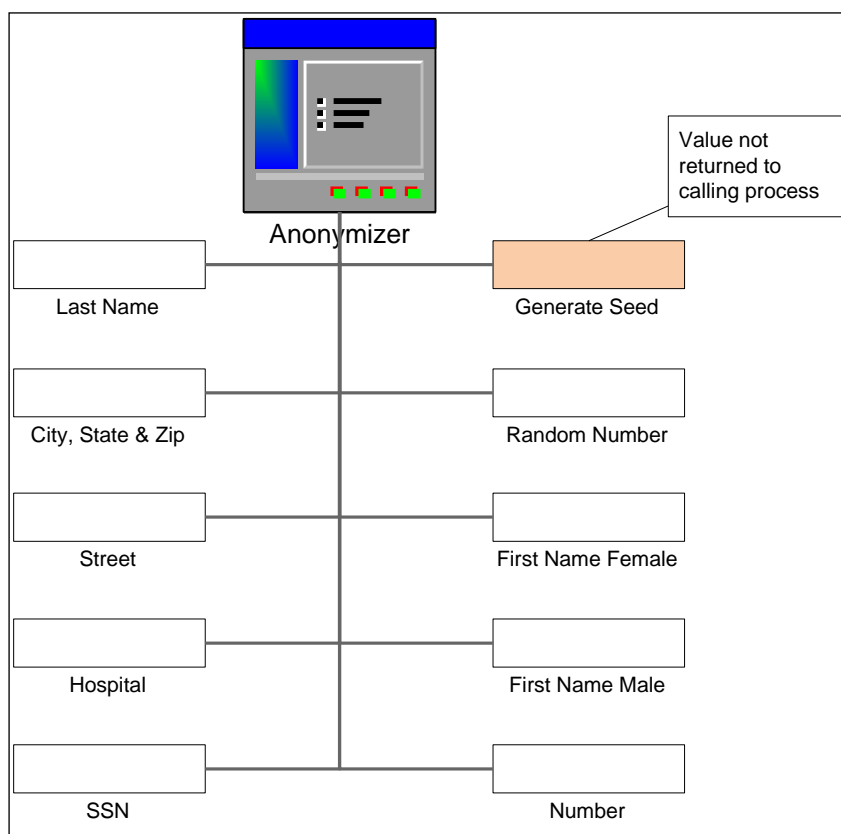
So how can we make the algorithm used a one-time event? Unlike a password file where we are trying to restore the original information (the password), in a medical data anonymization we want the source privacy information unidentifiable indefinitely. This allows us to work on a relatively unique hashing approach that is consistent for the one time conversion but not used in future conversions. That is, once the conversion takes place, the algorithm useful is discarded. Moreover, we need assurance that the algorithms used are cryptographically strong. Such a concept was posited in [Davis 1].

Before discussing the approach used, we will be discussing the use for the Secure Hash Algorithm (SHA-256). Readers may want to familiarize themselves with the SHA-256 [NIST 1]. In the case of the SHA-256, the National Institute of Standards and Technology (NIST) determined the algorithm is sufficiently strong for one-way conversions. However, as was previously mentioned, the algorithm is known so there needs to be an added capability to effectively make the algorithms unique.

In this paper, we use SHA-256 instead of a polynomial checksum (as described in [Davis 1]), the results depicted represent are based on randomly generated seed values. As such, every test run, where a new seed is created, will result in different results. The descriptions presented illustrate sample runs and by design, future runs using different seed values will present different results.

Approach

The anonymizer is a collection of programs that obfuscates meaningful information. The objective is to make it difficult to identify a single source from the anonymized information. At the same time, the converted information is in a format where research can be conducted and independently verified. We could model the anonymizer as an object with a series of methods (executable functions). For the initial phase, just the functional C-code is provided to verify the algorithm approaches presented.



Algorithm Description

The approach taken is to slightly modify the SHA-256 to allow the internal initialization to use a passed seed value. As long as the same seed value is used, every instance of the modified SHA-256 operates identically. In order for this to be useful, the seed value must be random and cryptographically strong. To do this we have one function to generate a seed value that is not returned (exposed) to the calling process. In so doing, the conversion is equivalent to a unique instance of the SHA-256. Consider, if we just applied the SHA-256, we could hash data until we found matching information and then extract the original information. In contrast, by seeding the process, we have no way to effectively determine where the SHA-256 started from. Consequently, there is no basis to hash text looking for the original

information. Consider hashing a social security number (SSN). There are 1 billion possible numbers and each could be hashed and placed in a sorted table. By knowing the initial starting point, every SSN could be recovered in real-time. In contrast, by not knowing where the SHA-256 started from, this type of attack is mitigated. For this reason, once calculated, the seed should be encapsulated at the processing point and not returned to the calling process. After the anonymization is complete, the seed should be destroyed by the anonymizer.

To re-emphasize the order of operation; the first step is to generate a seed value that will be used to convert a sensitive data source into a consistent anonymized result. As long as the same seed is used, multiple data sources should convert to the same anonymized data. However, to be effective, we need to take steps to increase the randomness of the seed value.

The seed function presented prompts the users to enter in a random string that is limited to 55 characters. If the user enters in more, the result is truncated. This value is then passed through the SHA-256 hash algorithm to produce 8 32-bit words (called the message digest). With this approach, we still have no assurance that the user will enter in a random string. If the string is known then a bad actor could recreate the seed value. To provide one additional step to help create a random number, the number of clock ticks it takes the user to enter in the sting is added to one of the initialization words and this value is hashed again. This is accomplished by adding the clock ticks value to S[2], the third number of the eight 32-bit message digest. By performing the second hash, using the last initialization array that includes the modification due to time, we have a reasonably random seed.

Two cautions here: 1) the code must ensure the seed value is purged completely after the complete set of anonymization is finished. 2) The user creating the seed should enter gibberish (meaningless string) when prompted. If these cautions are not followed, there is a risk that a known plain text attack could identify some of the initial starting material. Additionally, there are malicious software (Malware) programs that include keystroke capture that could aid bad actors in recreating the seed value. Knowing the keys entered should not compromise the web service (or server) that calculates the local clock ticks.

Why Modify SHA-256

In 2005, a statistical analysis was performed on the SHA-256 that examined 2-bit values, regardless of their location in the returned message digest [Davis 2]. That is, for each 256-bit SHA-256 message digests, there are 65,280 2-bit pairs. The analysis showed there were some weaknesses for the small block sizes used to generate the analysis. While the analysis does not explain where the weaknesses are, it does indicate there are certain values that exhibit slight weaknesses. Furthermore, the strength of the anonymizer is predicated on the strength of the hashing algorithm. To mitigate this vulnerability, the initial starting values are either the default specified by the SHA-256 or the seed value. By breaking out the starting value following a hash of the random text and then adding the time component to one of the 32-bit resulting values makes the algorithm different and stronger. A bad actor could not simple use a SHA-256 program, he or she would need to use the same algorithm presented.

SHA-256 modification

First, the SHA-256 requires an initial hash value specified in the table to the right.

At the start of the algorithm, the same values are used to initialize the algorithm. When the random character string is entered during seed creation, the initial values are used. However, once the hash of random characters entered is calculated, the

```
H[0] = (unsigned long)0x6a09e667;  
H[1] = (unsigned long)0xbb67ae85;  
H[2] = (unsigned long)0x3c6ef372;  
H[3] = (unsigned long)0xa54ff53a;  
H[4] = (unsigned long)0x510e527f;  
H[5] = (unsigned long)0x9b05688c;  
H[6] = (unsigned long)0x1f83d9ab;  
H[7] = (unsigned long)0x5be0cd19;
```

number of clock ticks is returned. Looking at the next code snippet, the computer clock() function, defined in <time.h> is used to provide the number of system clock ticks from the program start to the function call. That is, the time the program ran, to include the user's data entry is clocked and this value is then assigned to the variable time. In turn, the time is then bit-wise exclusive OR'd to the third hash word, H[2] (the third of eight). What this does is to include a time sequence that is difficult to calculate. This is in effect a second random sequence used in the seed creation. The slower the person entering the random text, the larger the returned clock() value. To ensure this is spread across the entire hash value, the string is hashed again using the modified initialization hash. This second hash ensures the time component is blended throughout the seed.

```
#include <time.h>  
#define SEED 1  
#define HASH 0  
long int time;  
time = clock();  
H[2] ^= time;
```

Within the create seed function, the call to the SHA-256 function uses three arguments. The first corresponds to the string, the second is the message digest (hash value), and the third is a flag to use the message digest argument instead of the standard default. A value of SEED means use the standard default. That is, a seed value is being created. This is compliant with the SHA-256 standard. However, when the value is HASH, the seed value previously created is used instead. This second approach is inconsistent with the NIST standard. However, it allows us to effectively create a unique variation that necessitates the use of the algorithm presented.

Alternatively, the message could be hashed and the clock added to the message digest. This in turn could be hashed and the resulting value would product a reasonably secure seed value. However, the approach taken is different enough that a bad actor would need to replicate the revised algorithm. To extract characters from the message digest, 8 bits are effectively shifted into a temp character. This then is converted to a number used to replace the original. That is, the 8-bit value is divided by 13 (the next largest whole integer greater than 12.8).

Seed Analysis

A test fixture was included to look at the seed generation process. Two runs are provided that detail what is taking place. In both cases, the same input string was used to show consistent hashing. That is, with the same string entered the same hash value results. However, the third number in the second value shows the addition of a time component. Finally, a new hash is taken using the derived seed as the new initialization. A third run shows the resulting hash is different when a new text string is entered.

Also note, the final string entered was truncated when the buffer length was reached (buffer overflow protection).

Please enter some random characters

The string entered was: **The quick brown fox jumped over the lazy dog's back.**

The hash of the random string entered

996569811 3233429169 **59377984** 3732065450 4180999389 1880185647 2191460990 918609726

The hash with time added to the third value

996569811 3233429169 **59392251** 3732065450 4180999389 1880185647 2191460990 918609726

The final hash is now the seed:

1759544386 2835325532 3652771055 417306058 2641065234 1024641131 993713972 2178881120

Please enter some random characters

The string entered was: **The quick brown fox jumped over the lazy dog's back.**

The hash of the random string entered

996569811 3233429169 **59377984** 3732065450 4180999389 1880185647 2191460990 918609726

The hash with time added to the third value

996569811 3233429169 **59395094** 3732065450 4180999389 1880185647 2191460990 918609726

The final hash is now the seed:

265698010 1893827635 3286510279 2659922043 4173743977 1949922236 3867031850 2333870658

Please enter some random characters

The string entered was: **Now is the time for all good men to come to the aid of**

The hash of the random string entered

1121722522 2097803415 **4025077855** 3076301616 2675321617 1183582577 886879685 4042325623

The hash with time added to the third value

1121722522 2097803415 **4025083536** 3076301616 2675321617 1183582577 886879685 4042325623

The final hash is now the seed:

1370050867 1371038584 1080987094 1189457279 1151625806 746625359 1319605404 4115996641

Social Security Number Anonymize

One item that has become a critical piece of information is the Social Security Number (SSN). This is used to identify individuals in multiple data sources. Furthermore, this is often used to exploit credit card information. As such, the SSN needs to be anonymized. Nine digits are used to represent the SSN meaning there are 1 billion possible combinations. The code snippet shows how the algorithm used. Essentially, the SHA256 is called to HASH the SSN. Note, the temp value starts as with the seed value and after hashing the SSN

returns the new message digest. Before hashing the SSN, the delimiters are removed from the original string. This is critical

```
copySeed(seed, temp);
tempNum = (unsigned long)atoi(ssn);
SHA256(ssn, temp, HASH);
tempNum = tempNum + temp[0]; /* using hash[0] */
tempNum = tempNum % 1000000000; /* get rid of the billions */
integerToString(tempNum, ssn);
```

because every instance of the SSN, which could be different amongst data sources, must be converted to the same format. For example, 012-34-5678 would result in a 012345678 value. If we hashed the SSN with delimiters, it would result in a value different from data sources using a different rendering. The SSN is depicted in two formats; a number and 9-digit character string. The character string is hashed and the first 32-bit value (temp[0]) of the resulting message digest is added to the SSN. Using modulo division, only the least significant 9-digits are preserved. In turn, these are converted back to a string format. Note: we use modulo division throughout to get indices into data tables. As a last step, the anonymized SSN is returned in the same delimited format in which it was received.

Number Anonymizer

The previous algorithm first converted the SSN into number before modifying. This approach provides a distribution across the 9-digits. However, there is often the need to anonymize numbers of varying lengths. For instance, the DOD-ID numbers designed to reduce the dependence on SSNs, is a 10-digit number. Although one digit is a checksum, the entire number is treated as a whole. Another number is the 11-digit benefits ID. This is often used to determine medical benefits and is another method for identifying the individual. Moreover, these different number schemes, like the SSN can have different delimiters to separate groups of digits.

Based on the previous discussion, a number anonymizer was prepared. The approach taken is different that SSN specific approach and tends to produce fewer 0's in the resulting number. That is, the algorithm converts each digit in sequence. To do this, the message digest of the original number (without any delimiters) is added to the digit and the result divided by 13. The resulting number is either a positive or negative number. If the number is negative, the number is inverted (thereby producing a positive number). Then the 9 and 0 digits are exchanged (normally, 9's occur less frequently because an 8-bit character does not divide evenly by 13. Note, normally we would expect the division to be by 12.8. However, we maintain the integer format resulting in a lower number of 9 results. Like the previous SSN anonymizer, this version maintains the delimiters passed. If there are no delimiters, then the returned value is limited to the number.

To examine how the numbers are skewed, consider the values 0 to 12, when divided by 13 result in a 0. Each group, except 9 and 0, has a range of 12. In the case of 0, the range is 13. In the case of 9, the number is 117 – 127 or a range of just 11. Given the message digest is essentially random, the distribution of numbers would normally be highest for 0's. Without knowledge of how the numbers are used, having a lead 0 could be problematic. To reduce the 0 frequency (while increasing the 9's) we swap the 0 with 9 digits.

Last Name Anonymize

Name	Frequency
Smith	1.006%
Johnson	0.810%
Williams	0.699%
Jones	0.621%
Brown	0.621%
Davis	0.480%
Miller	0.424%
Wilson	0.339%
Moore	0.312%
Taylor	0.311%
Total	5.623%

Table 1: Last Name Frequency

distribution of names. The question here is does this provide sufficient information to uncover the original last name or not? With larger random sets of names, more will translate to the same bucket as Smith. So while there is a strong correlation, the likelihood of absolute resolution is questionable.

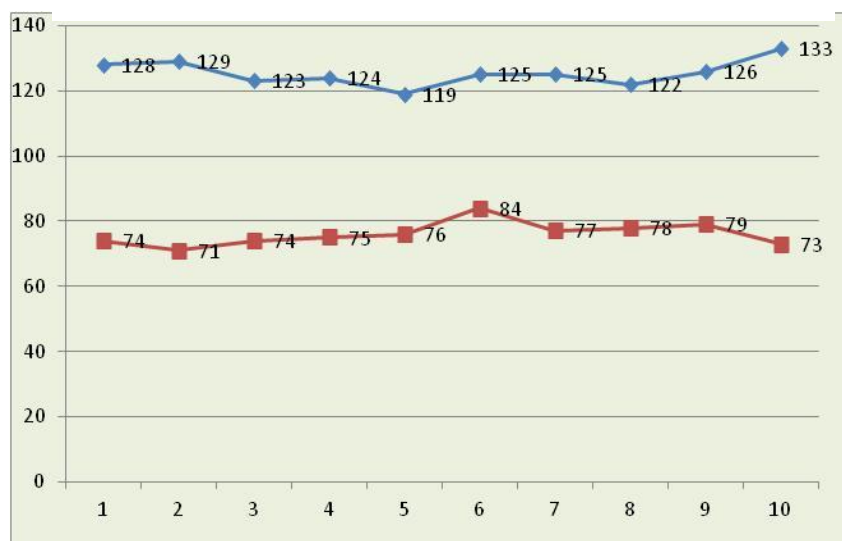
Next, the 10,000 most common names were used along with ten different seed values. Each of the 100 names was examined for the number of occurrences mapped. The next figure lists the maximum and minimum number of names that translated. For instance, the first run shows one name had only 74 translations while another had 128. Every other name translated to frequencies between these maximum and minimum values. Also note, the average is 100 (10,000 ÷ 100, that is 10,000 input names mapped onto 100 resulting names). So from this we see a frequency distribution that does not tend to fall into one name translation. Again, the goal is to anonymize the name such that uncovering the original is not certain. If we were to take the 10th run, were the maximum is 133, the next largest were 127, 117, 117, 116, and 115. That is, 133 names were translated into one. The question here is could we state the Smith last name absolutely mapped to the 133 translation or could it have mapped to one of the others?

For completeness, the standard deviations for the

10,000 most common names are included. It is important to note that the 10,000 most common names

The next function uses the algorithm posited to convert the total set of names into a smaller set of 100 names. What this does is consistently (using the same seed value) map to the same last name yet provide an ability to resist going backwards to recover the original name. If we look at the most common 10 last names, as depicted in the table to the left, we see that 5.623% of all names are one of these ten. If we attempted to use a statistical inference approach, we would expect 1% of all last names to be Smith. However, with only 100 last names mapped to, we would expect the top ten would have higher distributions. That is, for large random sets of last names, Smith would likely map to the highest

Figure 1: 10,000 Last Name Translations



were used only once. In practice, we would expect distributions to map those provided by the Census Bureau. Nevertheless, the distributions should provide sufficient ambiguity to mask most common last names. Especially since having the last name translated cannot be absolutely mapped back to the source name.

Anonymize First Name Male

The text anonymization functions work using the same core algorithms; albeit with different parts of the returned hash value.

The Census Bureau list of common first names for males is only 1219 names in length. Once again, the first 100 names are used in the anonymization process. Table 2 depicts a sample run showing the name in (bold) and the result out. Note, in this sample, Albert translated to Albert. For each conversion of the first 100 names, there is a 1% chance it will translate to itself. For the names beyond 100, there is no chance for a duplicate because the returned space is limited to 100 names. Note, names 101 – 1219 are never used in the anonymized results.

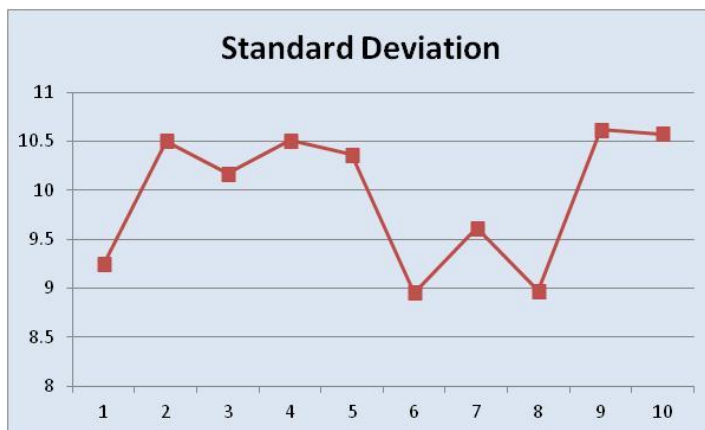


Table 2: Sample Translated Male First Names

In	Out	In	Out	In	Out	In	Out
James	Eric	Gary	Mark	Joe	Henry	Jeremy	Stephen
John	Peter	Timothy	Brian	Juan	Steven	Aaron	Martin
Robert	Roy	Jose	Jeremy	Jack	Albert	Randy	Lawrence
Michael	Andrew	Larry	Wayne	Albert	Albert	Howard	Roger
William	Daniel	Jeffrey	Robert	Jonathan	Willie	Eugene	Harry
David	Andrew	Frank	Juan	Justin	Anthony	Carlos	Edward
Richard	Joseph	Scott	George	Terry	Willie	Russell	George
Charles	Clarence	Eric	Edward	Gerald	Aaron	Bobby	Douglas
Joseph	Louis	Stephen	Ralph	Keith	Bruce	Victor	Joseph
Thomas	Jack	Andrew	Justin	Samuel	Lawrence	Martin	Johnny
Christopher	Benjamin	Raymond	Robert	Willie	Justin	Ernest	Earl
Daniel	Mark	Gregory	Jose	Ralph	Jack	Phillip	Wayne
Paul	Eric	Joshua	Peter	Lawrence	David	Todd	Kevin
Mark	Earl	Jerry	Earl	Nicholas	Harold	Jesse	Adam
Donald	Earl	Dennis	Michael	Roy	Juan	Craig	Gregory
George	Charles	Walter	Aaron	Benjamin	Carlos	Alan	Samuel
Kenneth	John	Patrick	Carl	Bruce	Joshua	Shawn	Craig
Steven	Bruce	Peter	Robert	Brandon	Jesse	Clarence	Bruce
Edward	Kenneth	Harold	Gerald	Adam	Thomas	Sean	Raymond
Brian	Brandon	Douglas	Frank	Harry	Steve	Philip	Patrick
Ronald	Richard	Henry	Larry	Fred	Lawrence	Chris	Jason
Anthony	Arthur	Carl	Charles	Wayne	Randy	Johnny	John

Kevin	Martin	Arthur	Ernest	Billy	Jimmy	Earl	Chris
Jason	John	Ryan	Jerry	Steve	Larry	Jimmy	Jonathan

The next figure shows the minimum, maximum, and standard deviation for the list of 1219 male first names. Note, due to the small sample size, the variances are more pronounced. Nevertheless, the results do show the names are converted according to the seed value selected. This also allowed the standard deviation to be displayed on the same graphic.

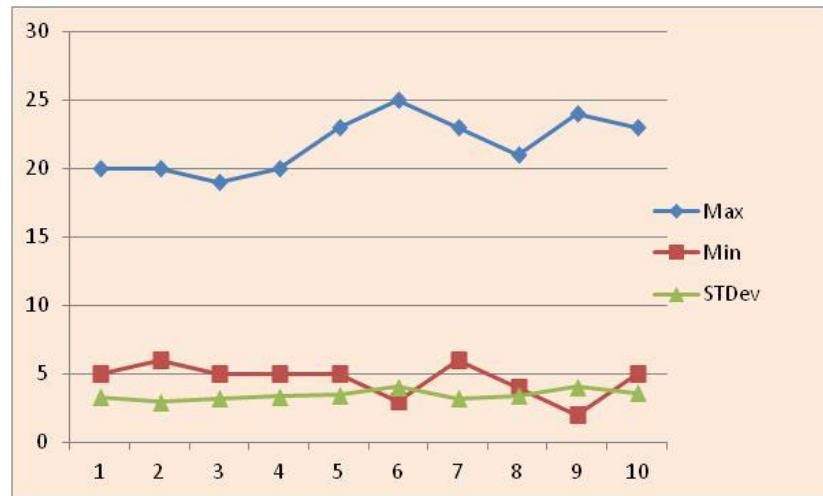


Figure 2: First Name Male Distribution

What this tells is that each of the 100 names was selected at

least twice and some were used up to 25 times during the sample runs. Again, the idea is to replace each name with one from the original set of 100. The anonymous names provide researchers with a method for referencing specific individuals without identifying the data source. As long as the same seed

value is used, all references to the anonymous person will be consistent. In Table 3, we see the distribution for the 10 most common first names.

Anonymize First Name Female

The Census provided a list of 4275 common female first names. The top 100 are depicted in the next table along with a single seed translation. In particular, we focus on the worse case translated information. Note four names, (Maria, Carol, Jane, and Lori) were translated to Carolyn. If we were able to use the same seed, the distribution, for large sets would have Lois (the translation of the most common name Mary) at greater than 2.6%. However, note that Christine (0.382%) and Kelly (0.283%) also translated to Lois. Thus the number of Lois first names for a large random set would be 3.294%. However, note that Elizabeth and Jennifer each have a frequency greater than 0.9% so the number of Mary's should be greater than 1.8%. What's more, the bulk of female first names lie outside of the top 10 most frequent names. By mapping to the limited set of 100 names, we are certain there will be multiple names overloaded. To determine if the translated name (Mary) belongs to Elizabeth or Jennifer would require other information.

First Name	Usage
James	3.318%
John	3.271%
Robert	3.143%
Michael	2.629%
William	2.451%
David	2.363%
Richard	1.703%
Charles	1.523%
Joseph	1.404%
Thomas	1.38%

Table 3: First Name Male Frequency

Table 4: Female First Name Translations

In	Out	In	Out	In	Out	In	Out
Mary	Lois	Jessica	Judy	Alice	Denise	Irene	Paula
Patricia	Heather	Shirley	Anne	Julie	Janice	Jane	Carolyn
Linda	Linda	Cynthia	Helen	Heather	Anne	Lori	Carolyn
Barbara	Cheryl	Angela	Sara	Teresa	Linda	Rachel	Michelle
Elizabeth	Mary	Melissa	Ruth	Doris	Ruth	Marilyn	Norma
Jennifer	Mary	Brenda	Brenda	Gloria	Virginia	Andrea	Anna
Maria	Carolyn	Amy	Emily	Evelyn	Deborah	Kathryn	Ann
Susan	Angela	Anna	Irene	Jean	Heather	Louise	Sarah
Margaret	Cynthia	Rebecca	Anna	Cheryl	Judy	Sara	Marilyn
Dorothy	Evelyn	Virginia	Christina	Mildred	Lisa	Anne	Heather
Lisa	Theresa	Kathleen	Marilyn	Katherine	Robin	Jacqueline	Amanda
Nancy	Anne	Pamela	Linda	Joan	Debra	Wanda	Jane
Karen	Irene	Martha	Gloria	Ashley	Shirley	Bonnie	Sarah
Betty	Heather	Debra	Gloria	Judith	Irene	Julia	Phyllis
Helen	Robin	Amanda	Lillian	Rose	Jane	Ruby	Louise
Sandra	Emily	Stephanie	Christina	Janice	Judy	Lois	Bonnie
Donna	Margaret	Carolyn	Jean	Kelly	Lois	Tina	Joan
Carol	Carolyn	Christine	Lois	Nicole	Julia	Phyllis	Dorothy
Ruth	Deborah	Marie	Wanda	Judy	Amy	Norma	Diana
Sharon	Sandra	Janet	Theresa	Christina	Judy	Paula	Julie
Michelle	Irene	Catherine	Beverly	Kathy	Joyce	Diana	Annie
Laura	Janet	Frances	Amanda	Theresa	Tina	Annie	Kathleen
Sarah	Elizabeth	Ann	Evelyn	Beverly	Michelle	Lillian	Michelle
Kimberly	Ann	Joyce	Barbara	Denise	Jane	Emily	Alice

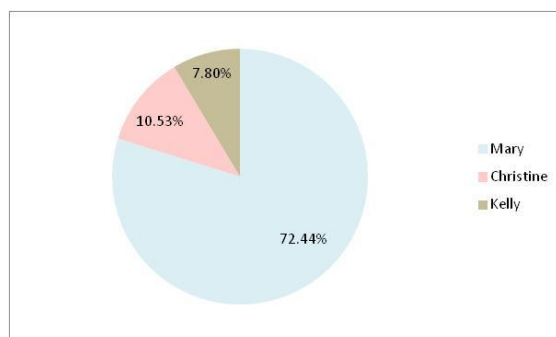


Figure 3: Mary Translation Example

If we look at the sample run distribution that translated to Lois, with the realization that this is but one random possible outcome, we can see how the most frequent name Mary translated. Here 72.44% of the translated names were Mary. However, the Christine and Kelly also translated to Lois providing a degree of uncertainty that is not easily resolved.

The question remains, does this offer sufficient obfuscation to effectively resist uncovering the source information? Part of the argument lies in the strength of the NIST SHA-256 algorithm that provides a relatively secure one-way function. So effectively, we are translating a first name into another name using one-way functions. Also note, the first and last names should be anonymized providing enhanced protection.

The next table depicts the frequency of the 10 most common female names (from the Census Bureau). Clearly, Mary is the most common name providing the worse case translation. That is, regardless of the seed used, for large data sets, there will be one name that has at least 2.629% of the translated names. If there is only one such name, we are certain that Mary translates to that name in question. However, we cannot be certain that the translated name actually is Mary. The reason is there is a finite set of 100 names that all names translate into. While there is a high probability the name was Mary, this is not certain. As seen from figure 3, in one sample run, the probability of the translated name being Mary was 0.7244.

First Name	Usage
Mary	2.629%
Patricia	1.073%
Linda	1.035%
Barbara	0.98%
Elizabeth	0.937%
Jennifer	0.932%
Maria	0.828%
Susan	0.794%
Margaret	0.768%
Dorothy	0.727%

Table 5: Frequency of Female First Names

Anonymize Streets

For the street anonymization, the approach taken was to remove the space between the text characters and convert to upper case before hashing. This is critical because any single bit difference will produce a different result. We should expect many addresses include extra spaces that would map differently. What's more, the census list of the most common streets contains 20 street names. Given this limitation, no effort was made to convert the ending street abbreviations (such as St., Cir., and Rd.). The algorithm is consistent with the name anonymization algorithms.

Random number generator

One common helper function is the random number generator. In actuality, the hash (message digest) provides a powerful random number. However, the need for a floating point number ($0 < x < 1.0$) finds its way into a number of math problems (including offsetting the date values). The random number generator is introduced.

```
float random(hashIndex)
    unsigned long hashIndex; /* use same structure for seed[index] */
    {
        float temp;
        temp = (float)hashIndex/(float)4294967296;
        return(temp);
    }
```

Here we use the calling function's unsigned long format for consistency. This number is then divided by the largest 32-bit integer and cast to a floating point. A short test fixture is used to depict sample random numbers.

```
main(argc, argv)
int  argc;
char *argv[];
{
    unsigned long seed[8];
    int index;

    createSeed(seed);
```

```

for(index=0; index<8; index++)
    printf("Random Number from seed[%d] = %f\n",index, random(seed[index]));
}

```

Note, this text fixture requires the external seed program and steps through each of the 32-bit values that comprise the message digest. A sample test run is shown in the text box. The important concept is that when the 2-bit seed values are determined by the source text hashed. As long as the same value is hashed, the same seeds are produced. Consequently, as long as the source date is consistent, the random number generated will also be consistent.

```

Random Number from seed[0] = 0.356295
Random Number from seed[1] = 0.896292
Random Number from seed[2] = 0.726034
Random Number from seed[3] = 0.513864
Random Number from seed[4] = 0.015362
Random Number from seed[5] = 0.516738
Random Number from seed[6] = 0.827733
Random Number from seed[7] = 0.063044

```

Hospitals

A function to change the name of the hospital is included. Every hospital is mapped into one of 20 possible hospitals with hypothetical names. The anonymized data would have a reference name for a hospital without having the actual location exposed. One reason for this is to ensure the anonymized information is not used for litigation against a specific facility.

City, State, and Zip code

It is expected these returned values will be used to replace the equivalent values in the calling process. The data used provides city, state, and zip values that are real. For example: Los Angeles, California instead of Los Angeles in another state. By converting to some reasonable location, the hope is relying applications can operate with minimum modification.

Note the index assignment is calculated by dividing by 42949673. This will provide a value between 0 and 99 inclusive. This can then be used as an array index where the translated information resides.

Also note, any city will map into only one out of 100 possibilities. So while there are thousands of American cities, only 100 will ever be used. This makes recovery of the original city difficult at best.

Another point, the source information could be multiple databases, data marts, or data warehouses. In turn, these could represent the information in different formats. For the anonymization to be consistent, the city string removes blanks and converts to upper case before hashing.

Dates

A date can be represented in a number of formats. Rather than craft code to accommodate every possibility, calling routines can use the random number generator to alter a numeric day of the month or numeric day. For example, by taking the random number and multiplying by 30 (then casting or truncating to an integer), a day of the month can be created. The same approach can be used for the month of the year. The random number can be multiplied by 12 and then cast to an integer. In both the

month and day, the value would need to be increased by 1 to shift the results to numbers starting at 1. For example, a day would return a range (0 through 29). By adding a 1, this shifts the result to (1 through 30). The same applies to months, adding a 1 the range changes from (0 through 11) to (1 through 12).

References

- [Davis 1] *Software Checking with the Auditor's Aid*, Proceedings of the Sixth Annual Computer Security Applications Conference, pp. 298-303.
- [Davis 2] Davis, Russell, SHA-256 Limited Statistical Analysis
<http://www.femto-second.com/papers/SHA256LimitedStatisticalAnalysis.pdf>.
- [NIST 1] National Institute of Standards and Technology (NIST) Secure Hash Standard, FIPS 180-3,
csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf.