CUDA Parallelized Obfuscation and Encryption for Data Analysis

COSC 89.25 Final Project Report

Jeff Liu, Matt Kenney, Andrw Yang

Background

Big data is often considered the 'oil' of the 21st century. Today, more than ever before, researchers and corporations alike rely on mining large datasets in order to drive insight, uncover patterns, and develop artificial intelligence algorithms which can improve the modern world. Unfortunately, big data analytics poses a serious legal and ethical risk when the data of interest is of a private or confidential nature.

Take medical analytics as an example. Medical researchers are often interested in conducting analyses such as genome-wide association studies (GWAS), in which they attempt to isolate a link between particular gene mutations and a disease phenotype. Conducting a GWAS study requires a large amount of computational power, and an extremely large quantity of human genetic sequences. Because of the extremely sensitive nature of human DNA sequences, GWAS researchers often experience difficulty in bringing their datasets to cloud computing environments, or in sharing their datasets with other researchers who would benefit significantly from the additional data. Privacy protection laws such as HIPPA add additional constraints on where and how GWAS researchers can use their datasets, preventing even the most careful and security-aware researchers from moving their data to another machine for analysis. Countless other medical research pursuits face this same issue. Conducting risk-factor analyses drawn from patient medical records, training machine learning models to read mammograms and distinguish between benign and malignant tumors, or analyzing latitude and longitude information of affected individuals to contact-trace disease contagion are all examples in which the datasets under analysis are extremely sensitive in nature.

The medical research community, however, is by no means the only user of sensitive data. Government researchers routinely use datasets marked as confidential (ex: US Census data), and also experience difficulty in moving their datasets from place to place. Cloud computing environments are often unsuitable for the kind of security that the government demands, forcing government researchers to rely on internal servers and computing environments.

Finally, the technology industry has amassed an enormous amount of consumer data, recording user preferences, personal information, web search history, health metrics (collected by smart watches and other similar devices), and more. Although big tech is often savvy enough to circumvent data protection laws such as GDPR by anonymizing datasets (i.e. removing

personally identifiable information), there remains a public outcry for tech companies to be more conservative with how they handle data collection, and tech companies too should be looking for ways to maintain the privacy of their users as they analyze the datasets they have amassed and sell them to other parties.

A variety of cryptographic techniques have emerged and, in recent years, matured, to address the issue of working with sensitive and legally-protected data. Each of these techniques is intended to accomplish the same task at hand -- to extract insight from a dataset, without ever revealing the raw data itself. Techniques such as federated learning and federated analysis primarily address the privacy concerns over personal consumer information, and attempt to draw insight from consumer data by training machine learning models and conducting standard analytical procedures locally on user devices (mobile phones, laptops, and desktops). Federated learning/analytics allows tech companies to draw insight from their consumers by collecting the results of analytical procedures run directly on user devices, rather than collecting raw consumer data itself. The technique of homomorphic encryption, alternatively, allows one to encrypt a dataset in such a way that one can run analysis on the encrypted data and obtain meaningful results without ever needing to decrypt the data whatsoever.

While the aforementioned strategies are attractive, however, they only work in certain cases. Federated learning only helps researchers to analyze distributed data such as consumer data, and remains a relatively undeveloped technology to date. Homomorphic encryption, while an astonishing innovation, is still unsuitable for big data analytics, often operating at 1 trillion times the runtime of traditional programs written to run on unencrypted data. The strategy that was utilized by this project -- obfuscation -- is able to be put into use *today*. Further, obfuscation offers a technically feasible, performant, and secure solution to performing analysis on encrypted datasets for most research use-cases.

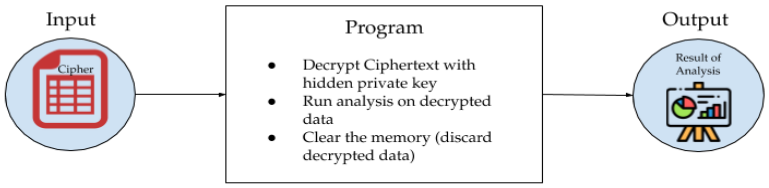
What is Obfuscation?

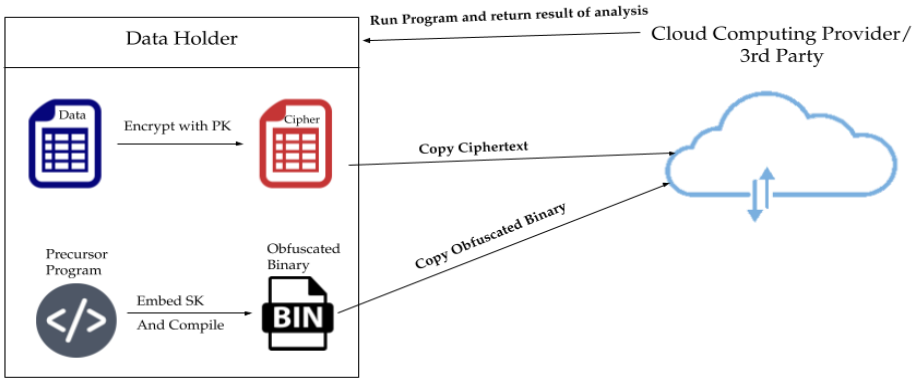
Here's how obfuscation works:

1. Generate a public/private key pair, or a single key in the case of asymmetric cryptography.
2. Write a program to conduct data analysis, machine learning, etc.
3. Embed the decryption technique and decryption key into the source code of this program.
4. Obfuscate the source code as much as possible without sacrificing too much performance.
5. Obfuscate the compilation process to generate complex and near-indecipherable machine code.
6. Send this compiled binary along with the encrypted data to the third party.
7. Run the code and pass the encrypted data into the program. Under the hood, the program:

a. Decrypts the passed dataset and places it in memory (never writing to hard drive).
b. Runs the desired analysis.
c. Clears the RAM such that the original data cannot be uncovered.
d. Returns the results of the analysis to the program user.

In this way, obfuscation allows for the analysis of encrypted datasets. Although it does not directly operate on encrypted data, by embedding the secret key deep into the program binary, and by never writing the decrypted data to storage, the obfuscated program keeps the data safe, all while allowing a researcher to gain valuable insight from that data.





Using this strategy, a researcher can now safely move *encrypted* versions of their datasets to a 3rd party cloud provider or share their encrypted datasets with other researchers. Researchers might even choose to share datasets encrypted with the same public key, aggregate these encrypted datasets into a single file, and utilize the same obfuscated program (containing the necessary secret key) to analyze their pooled data. One can imagine the flexibility and power a strategy such as obfuscation could provide researchers who are typically barred from collaboration and unable to leverage HPC resources in the cloud.

The Downsides of Obfuscation

Although Obfuscation is both performant, accessible, and widely applicable, it remains rarely used today due to concerns over the security of hiding a secret key inside of a program. Obfuscated codes are traditionally written in standard programming languages and built to run on standard processors (i.e. CPU). This comes with a variety of problems, namely:

1. There are a large number of experts on assembly code generated for CPU programs and for common programming languages. Assembly code experts may be able to examine obfuscated binaries and locate the hidden secret key.
2. Since CPU RAM is easily accessible, an attacker may be able to engage an attack which would allow them to "extract" decrypted data from system RAM in the middle of program execution.
3. CPUs have a number of documented hardware vulnerabilities (such as Spectre/Meltdown).

Several strategies have emerged to address these issues, one of the most interesting being the strategy of "Translingual Obfuscation." Translingual Obfuscation attempts to translate a standard program into an obscure programming paradigm. As reported in Wang et. al 2016,

> "[Translingual obfuscation is] a new software obfuscation scheme which makes programs obscure by 'misusing' the unique features of certain programming languages. Translingual obfuscation translates part of a program from its original language to another language which has a different programming paradigm and execution model, thus increasing program complexity and impeding reverse engineering."

Our Program: Obfuscation in CUDA

Through this project, we aim to expand upon the idea of "Translingual Obfuscation." Utilizing CUDA to implement an obfuscated program simultaneously allows us to introduce increased program complexity, due to CUDA's complex parallel programming paradigm, and to circumvent many of the problems with obfuscation on CPU. By utilizing CUDA, we are able to achieve a variety of security *and* performance improvements over standard CPU obfuscation:

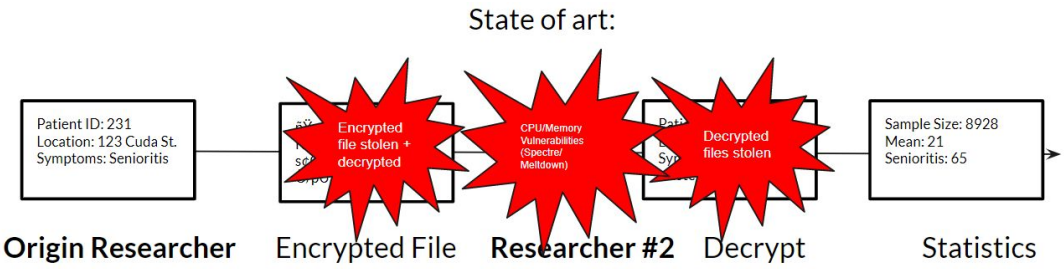
1. GPUs are good at decrypting massive amounts of data with parallel techniques
2. GPUs are now widely used for data analysis techniques (including machine learning) due to the massive speedup they provide when working on large datasets.
3. We hypothesize that uncovering a secret key that is held inside of an obfuscated GPU program is significantly more difficult than uncovering that same secret key embedded in a CPU program. GPU codes can be compiled for specific GPU architectures such that a hacker must own the GPU in question in order to obtain the assembly code. **Further, GPU assembly code is much harder to understand than CPU assembly code, and would require an expert on GPGPU assembly to decipher.** These constraints make it significantly less likely a hacker could extract the secret key from an obfuscated GPU program.
4. The added program complexity introduced by the parallel-programming paradigm will be unfamiliar to many adversaries, making the obfuscated binary even more difficult to crack.

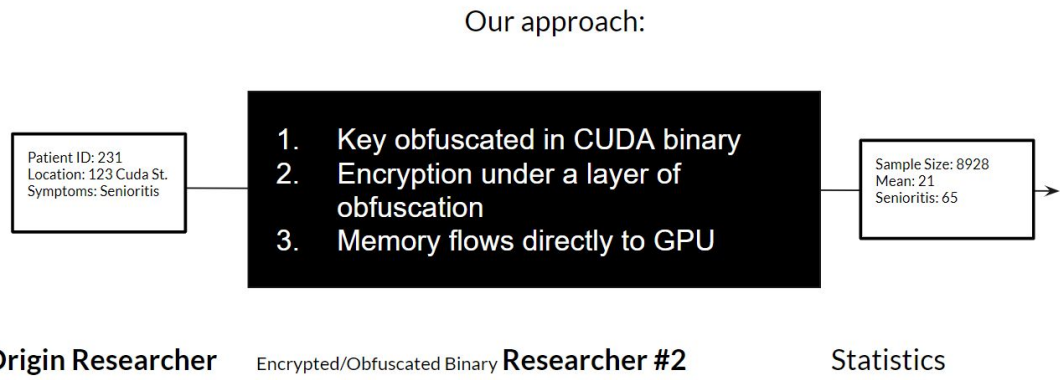
5. We store components of our secret key only in GPU device memory (specifically, we store it in device constant memory) making it more difficult to extract the secret key.
6. Decrypted data is stored exclusively on device memory, making it significantly more difficult for an adversary to obtain a copy of the decrypted data than it would be if the data were stored in standard system RAM.

Traditional CPU obfuscation is vulnerable to binary cracking, especially because CPU assembly is well-studied and documented, allowing adversaries to potentially extract a hidden secret key. In particular, to circumvent this issue and hide our secret key using parallel programing paradigms, our program does not store the value of the secret key but rather stores the key in distributed parts. We split our key before it is hardcoded into the binary. Utilizing matrix multiplication, we are able to store the key across 513 values stored in our algorithm, and multiply these components together to achieve a resultant matrix which contains the secret key. We will discuss this obfuscation strategy in detail further along in this report. With CUDA, we are able to achieve this matrix multiplication in a parallelized fashion. A further implementation of our strategy could distribute the secret key in far more than 513 discrete values for further security. We implement parallelization throughout the program both as a HPC method as well as an obscurity factor.

Our program diminishes traditional hardware vulnerabilities by keeping the decrypted data entirely on the GPU across multiple blocks/grids. The decrypted data will never exist on system RAM/CPU. CPUvulnerabilities, such as Spectre and Meltdown, will not be able to put the data at risk. *By avoiding the host side of the computer almost entirely, we greatly diminish the attack surface.*

Throughout this program we have two goals: first to parallelize security elements (obfuscation and encryption), and second to run a statistical test in a HPC setting. Thus, our primary criteria is avoiding host-side memory and CPU functions with time performance being a secondary benefit. Our program is primarily intended as a proof-of-concept for building obfuscated programs in GPGPU frameworks, and our work reflects this intent.





Implementation

Our program consists of many modules, each of which work together to compose the obfuscated analysis system in its entirety.

External Modules (outside of the obfuscated binary):

1. A simple key-pair generation script (`keygen.py`) . This python script generates a 64-bit (unsigned long long) RSA, consisting of three 64-bit numbers - e, n, and d. Collectively, (e,n) represents the public key, and (d) represents the private key. In addition to generating the necessary encryption/decryption keys, this script generates the 513 values which we use to reconstruct the private (or "secret") key within the obfuscated CUDA code.
2. An encryption script (see the `Encryption-CSharp` directory) written in C-sharp. This script encrypts and pads (more detail under "Encryption/Decryption" section below) a dataset using the RSA encryption key generated by `keygen.py` and stores that dataset as a binary file.

CUDA Modules (contained within the obfuscated binary itself):

1. A CPU code to map the encrypted binary file representing the sensitive dataset directly into RAM .
2. A GPU kernel code which computes the secret key and stores it in device memory (more detail under "Obfuscation" section below).

3. A GPU kernel function used to transpose the encrypted data (originally stored in CSV format) into column-major order for faster data processing and analysis.
4. A GPU kernel function to decrypt and depad the encrypted data and store the decrypted data in device global memory.
5. A GPU thrust module which operates on the decrypted data to calculate a simple statistical test and returns the results of that statistical test to the end-user.

We describe the most important of these modules below.

Encryption/Decryption Design and Schema

Due to constraints within CUDA, we had to design our own bit-level RSA encryption scheme. Traditional RSA cryptosystems utilize 1024-2048+ bit integer key values. Unfortunately, CUDA does not natively support 1024-2048 bit integers, especially across devices of older architecture. Although a production-grade obfuscated CUDA code would need to employ 2048+ bit keys, we chose to use unsigned 64 bit integers as a proof of concept. As GPU libraries designed to work with arbitrarily large numbers (CGBN) and GPU technology advance, implementing higher bit encryption schemas will be possible. We used the prime number functions `modexp()` and `modmult()` to carry out our encryption and decryption. Beyond data choices, we designed a padding schema to pad data before encrypting (a standard process in RSA encryption).

```
data[col] = modexp(cipher[col], d[0], n) & 0x00000000ffffffff;
```

In order to prevent data points with the same value in the original dataset from taking on the same value in the encrypted dataset, we needed to inject random bits ("pad") to avoid similar encrypted data points. We injected random integer values between the 6th and 7th most significant bytes of the data. Our 8th (most significant) byte was left as a zero value in order to prevent our encrypted data from becoming larger than the RSA key size itself (RSA requires that input data has a smaller bit length than the RSA key). This preserved data variability for encryption (A more thorough and future implementation of padding would be full Optimal asymmetric encryption padding, or OAEP). For decryption and depadding, we used a bit mask (0x00000000ffffffff) and the bitwise AND operation to "depad" the data.

```
gpu_decrypt<<<num_blocks, blockdim>>>(dep_cipher_gpu, dep_data_gpu, numrows, d, n);
```

Parallelization was used for both encryption and decryption. In terms of encryption, we used TPL in C# to achieve approximately a 2x speedup. For decryption, we wrote a kernel function to operate on a matrix with transpose data. We transposed the data for more efficient memory accesses during this step. By taking the transpose before decrypting, we noticed a 100-200x speed up in decrypting on the GPU vs. decrypting on the CPU. *Beyond speed, the major benefit of parallelization is that our unencrypted and sensitive data never leaves the GPU.*

*For documentation, our encryption used Microsoft's TPL (similar to OpenMP) to achieve a 2.8s to 1.3s speed up for raw data encryption
**A limitation of our implementation is that data size is limited to 5 bytes per value

Obfuscation

One of the most fundamental and basic rules of cryptography is to never hardcode your secret keys in plain text. A typical cryptographic system does not need to worry about this rule, since the key is kept secret quite separately from the decryption schema. However, since we intend to hide the key within the program such that not even the person running the program can see it, we need to keep the key secure through obfuscation. We do this by using parallel algorithms to calculate the key upon runtime.

Having generated the key using traditional RSA key generation algorithms separately from the CUDA program, we used a Python program to convert that key into two 16 by 16 matrices and an adjustment value. These matrices had semi-random values evenly distributed around an arbitrarily selected mean. In the Python program, we multiplied the matrices, performed a reduction sum on all the values in the resultant matrix, and compared that sum to the key. The difference of those is the offset value.

```
import numpy as np
# generates a sidelen x sidelen random normal matrix with
# values normally distributed around base_value
matrix1 = np.random.normal(mean, sd, (sidelen, sidelen)).astype(int)
matrix2 = np.random.normal(mean, sd, (sidelen, sidelen)).astype(int)
result = np.matmul(matrix1, matrix2).sum()
adjustment = pk - result # offset value
```

These two matrices were then embedded into the source code of the CUDA program as 512 values (two matrices containing 256 values) stored in constant device memory. To regenerate the key in device memory at runtime, the CUDA program would calculate the matrix multiplication of the two stored matrices. Then, it would use parallel reduction techniques to sum all the values and add the precalculated offset constant. The resultant secret key would be stored on device memory to prevent attacks on the CPU from compromising the key while the program is still running.

```
__device__  __constant__ unsigned ll m1[16*16] = {...}
MM_Sum<<<1,dim3(16,16)>>>(m1_raw,m2_raw,key_dev,16,16,16);
```

These 513 constants can be hardcoded into the source code with greatly reduced risk. It is difficult to reverse-engineer a matrix multiplication programmed with parallel techniques. However, this technique is not yet perfect. To create a more securely obfuscated key, we can multiply much larger matrices, or perform operations that are similar to a matrix multiplication

but slightly different in a way that is indecipherable to a hacker. That way, even if an adversary mined the entire set of matrix values and the offset value from the assembly code and also knew that we used matrix multiplication, they may not be able to reproduce the exact method we use to calculate the secret key.

Statistics

In a real-world application, sensitive datasets are full of incredible data that can be analyzed. Various machine learning algorithms can be employed -- from multiple regressions, to principal component analysis, to deep learning -- all of which need heavy yet massively parallelizable computation. Much of modern data science already runs on GPU clusters, and it is natural to use a GPU to improve our statistical time performance.

However, being undergrads, some of our team members were not well-versed in the state of the art in data science. In addition, the focus of this project is on the obfuscated decryption. After that is accomplished, it is trivial to add more advanced data science algorithms, since the correct data has already been protected and selectively decrypted. For these reasons, we opted to do comparatively simple statistical inference tests instead. The statistics in this project primarily serve to demonstrate that the data returns insightful results while staying encrypted. The statistics themselves are not the focus of our work.

In order to test the functionality of our obfuscated program, we chose to analyze data collected by the Korea Centers for Disease Control & Prevention (KCDC) on patients who had tested positive for COVID-19. We examined this dataset of COVID-19 patients in Korea and tested for differences between Male and Female mortality using the Student's T-interval and Welch's T-test for different variances. These tests both rely on finding the mean and standard deviation of datasets, then calculating various summary statistics that can be compared to the cumulative distribution function of the T-distribution to return insight. For the CPU, we implemented these calculations with for-loops, and on the GPU, we used Thrust to perform parallel reduction, along with custom functors(function objects) to calculate standard deviation.

```
variance[0] = thrust::transform_reduce(thrust::cuda::par.on(s1), data.begin(), data.begin()
          + length[0], std_dev_func(mean[0]), (double)0, thrust::plus<double>());
```

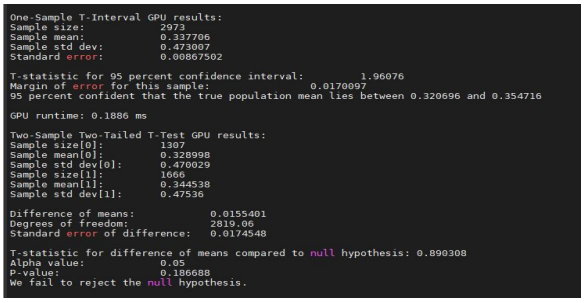
This type of reduction on some datasets is not easily parallelizable, and we saw a decrease in performance on the GPU. Particularly, it can be difficult to reduce mean and standard deviations for different categories. We also used a stopgap solution of parallel sorting of the data, which is a slowdown. In addition, we attempted to use Thrust's execution policy header to call Thrust kernels asynchronously on multiple streams, but had little to no speedup. More

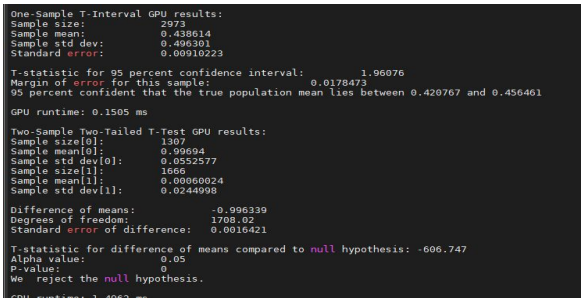
advanced SIMD paradigms for sorted reduction are possible, but since the focus of the project is on decryption and obfuscation, our implementation suffices for T-tests.

An important thing to note is that we cannot merely handle statistics work on CPU when certain calculations are marginally faster there. One of our objectives is to hide data from the CPU and keep as much of it as possible on the GPU to evade RAM siphoning attacks. Thrust operates on device vectors, which safely keeps the data on the device without copying any values to the host.

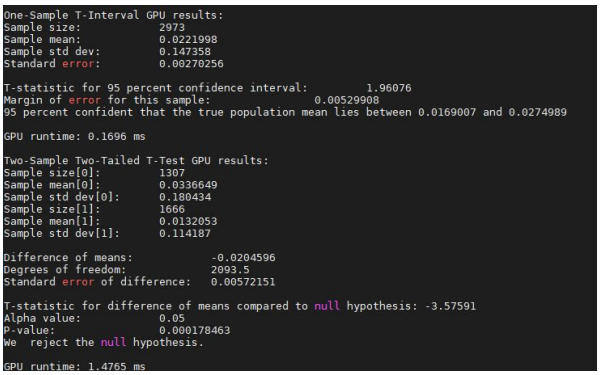
Testing and Results

To test the accuracy, results, and insight gained from the program, we created two fake datasets and also analyzed a third, real-world dataset. The first two data sets had either extreme correlation between gender and mortality rate or a randomly generated distribution between gender and mortality rate. The third dataset was pulled from KCDC data on Korean patients near the beginning of the outbreak.





The above analysis results were done for the uncorrelated sample on the left, and the correlated one on the right. We get results exactly as expected, indicating that our program is correctly calculating the key, decrypting, and depadding the data, and giving us correct statistical results.



This analysis contains the results from the real KCDC data. It rejects the null hypothesis and indicates statistically significant evidence that males suffer a higher mortality rate than females.

This conclusion is corroborated by the current medical consensus on the association between gender and COVID-19 morbidity and mortality, which points to the accuracy of our results. In addition, we can see from the confidence interval that we can be 95% sure that the overall mortality rate during the early phases of infection in Korea is between 1.7% and 2.7%, which is also consistent with current medical research.

Performance

We created two versions of this program--one done entirely on the CPU, and one done entirely on the GPU. The CPU solution is not exactly a "true" corollary to our GPU solution since it lacks most of the security features that we use CUDA for. However, it serves as a decent benchmark to which we can compare our parallelism speedup. The summary results for our timing benchmarks with real KCDC data are below:

	CPU	**GPU**
Decryption	500ms	2.7ms
Statistics	0.13ms	0.25ms
Total	500.13ms	2.95ms
Security	Encrypted	End to End Encrypted, Obfuscated

The biggest speedup by far was found on our decryption module. Since the decryption of each data point is completely independent of any other data points, this instruction is fully parallelizable. Single-threaded decryption simply isn't a good technique to process thousands of lines of data. In addition, data transposition from a row-major CSV-like cipher file to column-major data access benefits from GPU parallelization. We experienced a speedup of almost 200 times on decryption. With more advanced digital arithmetic functions and multi-threaded CPU execution, this gap may be narrowed. Regardless, the GPU implementation will still be much faster and also much more secure.

As mentioned earlier, our basic statistics were already very fast on the CPU, and were not the bottleneck of this program. The GPU parallelism was not great for calculating simple means and standard deviations, especially for heterogeneous categorical data. However, as mentioned earlier, the T-tests are a proof of concept, and their performance is not the focus of this project. In addition, when dealing with sensitive data, an increase in security is always worth a small decrease in speed.

In a real-world application, far more advanced data science methods would be used. Virtually all modern machine learning techniques (and many other statistical procedures) run orders of magnitude faster on the GPU. If we had implemented some of those algorithms, we would see a huge speedup there as well.

Compiling and Running the Code

- Although this step is unnecessary since we have already stored all private key components within our obfuscated CUDA code, one could generate the public/private key pair along with the the matrices/offset used to calculate the private key by installing the python libraries `gmpy` and `numpy`, running `python keygen.py 64` (64 representing the desired RSA key length), and viewing the results in the resultant `key.txt` file. Note that the `keygen.py` file is located in the `Cuda` directory.

- To generate the encrypted data, one can build the C# code contained in the `Encryption-CSharp` directory. One can then pass the CSV file representing the dataset of interest to the resultant executable. Using Visual Studio 2019 tools, the encrypting binary can be built with: `devenv /build Release Encryption-CSharp.sln` or `csc Program.cs` The original South Korea COVID-19 data is stored in the `Data/RAW_PatientInfo.csv`, and the doctored versions of this dataset are stored in `Data/ hypercorrelated.csv` and `Data/uncorrelated.csv`. These files can be imputed as the first argument into the encryption binary. This C# program will generate binary encrypted files with a `.dat` file descriptor.

- We have created a CUDA program purely for demonstration purposes which notifies the user of the results at every step of the obfuscation process (i.e. shows the user what the encrypted data looks like, what the decrypted data looks like, and the results of the statistical analysis). This program is, of course, not suitable for true obfuscation, but is quite useful as a demonstration of how the underlying obfuscation process actually works. This code also runs the obfuscation process on both CPU (storing the private key directly as a constant) and GPU (where the private key is distributed into 513 values) in order to test the performance of our code. To compile this code, simply run `nvcc obfuscated_prog.cu -o obfus`. You can then execute this code by running `./obfus ../Data/orig_data.dat` to conduct an analysis on the original South Korea COVID data, or you can pass `hypercorrelated.dat` or `uncorrelated.dat` to conduct analyses on the doctored COVID datasets. The code should be run with option `-std=c++11` if nvcc does not default to C++11.

- We have also created a CUDA program that runs the obfuscation process and data analysis only on GPU, and serves to represent the intended behavior of our obfuscated

program. This program takes an encrypted dataset as input, and outputs only the result of the computed test statistic. To compile it, simply run `nvcc gpu_only_obfuscator.cu -o obfus` and run the program as described above.

Member Contributions

Data Schema	Andrw, Matt, Jeff
Encryption	Andrw
Decryption	Matt
Statistics	Jeff
Combining/Debugging Kernel Functions	Andrw, Jeff, Matt
Obfuscation	Jeff, Matt
General Debugging	Matt, Jeff, Andrw

NOTE: Andrw started his internship on June 1st, as a result, most of his contributions were front loaded in the project.