Data Oblivious Genome Variants Search on Intel SGX

Avradip Mandal[1], John C. Mitchell[2], Hart Montgomery[1], and Arnab Roy[1]

[1] Fujitsu Laboratories of America, Sunnyvale, CA, USA
[2] Stanford University, Stanford, CA, USA

Abstract. We show how to build a practical, private data oblivious genome variants search using Intel SGX. More precisely, we consider the problem posed in Track 2 of the iDash Privacy and Security Workshop 2017 competition, which was to search for variants with high χ^2 statistic among certain genetic data over two populations. The winning solution of this iDash competition (developed by Carpov and Tortech) is extremely efficient, but not memory oblivious, which potentially made it vulnerable to a whole host of memory- and cache-based side channel attacks on SGX. In this paper, we adapt a framework in which we can exactly quantify this leakage. We provide a memory oblivious implementation with reasonable information leakage at the cost of some efficiency. Our solution is roughly an order of magnitude slower than the non-memory oblivious implementation, but still practical and much more efficient than naive memory-oblivious solutions–it solves the iDash problem in approximately 5 minutes. In order to do this, we develop novel definitions and models for oblivious dictionary merging, which may be of independent theoretical interest.

1 Introduction

A *trusted execution environment* (TEE) is a secure area of a main processor. In particular, a TEE attempts to simulate a 'black box' environment: users (even with physical access) of the main processor may only see the inputs to and outputs from the TEE, and learn nothing about the data or processes inside the TEE. This 'black box' premise potentially allows for private, secure distributed or cloud-based computations on data that previously were only known to be possible from very heavyweight, impractical cryptography (or even not known to be possible!).

Examples of TEEs available today include Intel's SGX (Software Guard Extensions), ARM's TrustZone, AMD's Secure Execution Environment, and Apple's Secure Enclave. There are many different types of TEE in existence today, but in this work we will focus on SGX, which is currently the most studied TEE.

TEEs are particularly exciting for applications where we want third parties to perform computations on secret data. For instance, if we assume a secure TEE, it is known how to build many powerful cryptographic primitives that run with very small overhead when compared to native computations: fully homomorphic encryption [SCF+15], functional encryption [FVBG17], and even obfuscation [NFR+17] are all known to be practical with trusted hardware. Coupled with other cryptographic techniques, these primitives implicitly allow a vast range of functionality for TEEs: things like secure linux containers [ATG+16], oblivious multi-party machine learning [OSF+16], and blockchain smart contract messaging techniques [ZCC+16] are possible and efficient when TEEs are used.

iDash Competition. To further illustrate the power of TEEs, consider the following scenario: suppose a medical research institution wants to outsource aggregation and statistical computation on genome data to a TEE based cloud server. Individuals would send their encrypted genome to the cloud server. TEE would decrypt the encrypted data, perform statistical computation and send back the end result to the research institute. With traditional cryptography, to achieve comparable security we would require functional encryption for very complicated functions, which is only known from indistinguisihability obfuscation (and is extremely inefficient). This scenario almost exactly describes the 'track 2' problem given in this past year's iDash competition [iDa17], which is a privacy and security workshop devoted to using cryptographic techniques to help solve problems in computational biology and genetics. In Section 3 we will describe the problem in details. Among proposed solution, the best solution was due to Carpov and Tortech [CT18], which performed the computation on 27.4 GB of data in only 65 seconds of client-side preprocessing time and 7 seconds of enclave time.

Side Channels. Unfortunately, it is easy to see that there are many ways a potential adversary can learn about computations in the TEE–even if the TEE is 'perfectly' secure, as long as it has finite computational power, finite memory, and connections to other outside systems, there are ways for an adversary to learn things about secret information. For instance, an adversary could measure the time that a particular computation takes and use that to infer things about secret information involved in the computation. Often, the TEE does not have enough internal memory to store all of the data needed for a particular computation. In this case, it must store (encrypted) data in outside locations, like regular memory or hard disks. When this happens, an adversary can observe the memory access patterns of the program running inside the TEE and also potentially learn secret information.

These kinds of atttacks are called *side channel* attacks and have been widely known in the cryptographic community since Paul Kocher's famous paper [Koc96] which long predates modern trusted hardware. The history of side channel attacks include things like observing how long a computation takes [Koc96], tracking the memory access patterns of a particular program [KSWH98,Pag02], and measuring power consumption at given times when the program is run [MDS99].

Side channel attacks on SGX and other TEEs have been proposed for almost as long as the TEEs themselves have existed. Most of the side channel attacks on SGX have focused on the cache [GESM17,BMD+17]–in other words, the lack of ‘memory obliviousness’ of programs–but there have been other side channel attacks, including attacks based on timing [WKPK16]. In addition, there has been a lot of research done with the goal of mitigating these side channel attacks. Many techniques, like oblivious RAM (ORAM) [Gol87] or path ORAM [SvDS$^+$13b] are very general and can do a lot to mitigate these side channel attacks. In fact, there has been quite a bit of research lately on preventing certain classes of side channel attacks in SGX [SLKP17,SCNS16,SLK$^+$17]. Unfortunately, the generality of many of these techniques typically implies a large overhead, and thus the resulting TEE-based schemes are not very efficient. Oblivious B+ tree implementations using shuffle index are also well known [VFP$^+$15]. However, as described in Section 3.1, in our context the optimal data structure is dictionary or hash table.

Our Contributions. Like many other SGX-based protocols, all of the submissions in ‘Track 2’ of the past year’s iDash competition were potentially vulnerable to side channel attacks. In this paper, we show how to build a provably side channel resistant variant of the fastest (and winning) submission [CT18]. We employ a number of techniques, including oblivious shuffles and dictionary merging, as well as clever cache management, in order to provide provable resistance to side channel attacks.

While our side channel resistant construction massively outperforms what generic solutions like ORAM would give, it is still not quite as efficient as the native solution in [CT18]. While the solution of [CT18] takes 65 seconds of preprocessing time and 7 seconds of enclave computation time, our memory oblivious solution which only leaks aggregate intersection sizes among input data (see Section 6 for details) takes 28 seconds of preprocessing time and about 5 minutes of enclave computation time–significantly less efficient than the non-memory oblivious solution, but certainly practical.

In order to achieve memory obliviousness, we construct new definitions and models for oblivious dictionary merging. These models help us to formally state properties about memory obliviousness and may be of independent theoretical interest.

Outline. The rest of the paper is as follows: in Section 2, we discuss the security model we use around SGX. We next define the genomic search problem from the iDash Track 2 that we have alluded to earlier in Section 3. We also explain the (non-side channel resistant) winning solution in this section. In Section 4, we discuss how to make the previously discussed solution memory oblivious (and thus, side channel resistant). Then, in Section 5, we discuss how to merge dictionaries in a memory-oblivious way, which is a critical component for our overall solution. We discuss our experimental results in Section 6.

2 Security Model of SGX

A program is called data oblivious if its memory access trace can be simulated by a simulator with access to only some observable information. In theory, one can use Oblivious RAM implementations to make a program data oblivious. However, generic application of ORAM [SVDS+13a] techniques with small amount of trusted memory has a large overhead, compared to native running time. But what is trusted memory inside an SGX enclave? A conservative approach might be to consider only the CPU registers as trusted memory. On the other end of the spectrum, an optimistic approach can assume all available enclave memory (about 96 MB) as trusted memory. Taking this optimistic approach authors in [EZ17,ZDB+17] showed many SQL like database operations can be performed in an data oblivious manner with very little performance overhead.

A reasonable model of trusted memory lies somewhere in between. All data in the Last Level Cache (LLC) remain unencrypted. So it's quite natural to assume the LLC is part of the trusted memory. However, the size of the LLC available to the enclave program is controlled by the adversary with a 4KB (cache line) granularity.

To be reasonably conservative, in this paper we assume that all memory accesses are visible to the adversary. In particular, we follow the model of Chan et al. [CGLS18], who introduced the notion of adaptive strongly oblivious simulation security for arbitrary stateless functionalities and Oblivious Random Access Machines (ORAM). Given a stateless functionality f, some leakage function $\texttt{leakage}_f$, $\texttt{Alg}_f$ obliviously implements f with leakage $\texttt{leakage}_f$ if

- $\texttt{Alg}_f$ correctly computes the same function f except with negligible probability for all inputs,
- the sequence of addresses requested (and whether each request is read or write) by $\texttt{Alg}_f$ do not reveal more information than the allowed leakage.

Formally,

Definition 1. *$\texttt{Alg}_f$, obliviously implements the functionality f with leakage function $\texttt{leakage}_f$, iff there exists a p.p.t. simulator $\texttt{Sim}$, such that for any non-uniform p.p.t. adversary $\mathcal{A}$, $\mathcal{A}$'s view in the following two experiments are indistinguishable or equivalently $\| \Pr[b_{real} = 1] - \Pr[b_{sim} = 1] \|$ is negligible in terms of security parameter λ.*

Algorithm 1 Real Experiment
```
procedure $\textsc{Expt}_{\mathcal{A}}^{\texttt{REAL},\textsc{Alg}_f}(1^\lambda)$
    $\mathcal{A} \to I$
    $\texttt{out}, \texttt{addresses} \leftarrow \texttt{Alg}_f(I)$
    $\mathcal{A}(\texttt{out}, \texttt{addresses}) \to b_{\texttt{real}} \in \{0,1\}$
end procedure
```

Algorithm 2 Simulated Experiment
```
procedure $\textsc{Expt}_{\mathcal{A}}^{\texttt{SIM},\textsc{Alg}_f}(1^\lambda)$
    $\mathcal{A} \to I$
    $\texttt{out} \leftarrow f(I)$
    $\texttt{addresses} \leftarrow \texttt{Sim}(\texttt{leakage}_f(I))$
    $\mathcal{A}(\texttt{out}, \texttt{addresses}) \to b_{\texttt{sim}} \in \{0,1\}$
end procedure
```

Here $\texttt{addresses}$ in the real experiment denotes the sequence of addresses requested by $\texttt{Alg}_f$ along with the information whether each access is read or write.

Case Study: Oblivious Sort. Traditional implementations of sort typically proceed by repetitively comparing two values and swapping them or doing nothing depending on the result of the comparison. This induces them to produce different access patterns based on the data values themselves, and as such they are not oblivious. However, some algorithms such as bitonic sorting [Bat68] are data independent, and hence oblivious. In addition, “swap or not”-based sorting algorithms can be made oblivious by accessing the same memory locations regardless of the comparison outcome. In Section 4, we will use oblivious sort primitives for oblivious implementation of genome variants search.

Case Study: Oblivious Shuffle. Oblivious shuffle is a simple but important stateless oblivious primitive. As the name suggests, the shuffle algorithm takes a sequence of n elements as input and outputs a uniformly random permutation of the sequence. Consequently, an oblivious shuffle is an algorithm whose memory accesses can be simulated irrespective of the input and the output, and hence also the actual permutation that was employed. A natural way to do an oblivious shuffle is to pair each entry with a uniformly random number and then oblivious sort the pairs with respect to the random numbers. Other efficient algorithms which are not based on sorting also exist [OGTU14]. In Section 5, we will use oblivious shuffle primitives for realizing oblivious dictionary merging.

3 Whole Genome Variants Search

In this section we provide a very short introduction to genomics and describe the Genome Variants Search algorithm which identifies genes responsible for certain hereditary diseases.

DNA (Deoxyribonucleic acid) is a chain of nucleotides with the shape of a double helix. It carries genetic information in all living organisms. The complete genetic material of an organism is called its genome, and DNA is identical in every cell of our body. A very long DNA chain forms what is called a chromosome. Humans have 23 pairs of chromosomes, and each pair has one chromosome from the person’s father and one from the mother. Any two humans share about 99.9% of their DNA. The remaining 0.1% DNA tracks the difference between two individuals. Most of these differences occur in the form of what is called a Single Nucleotide Polymorphism (SNP). A SNP is a variation in a single nucleotide that occurs at a specific position in the genome (compared to a reference genome). Moreover, a SNP can be either heterozygous or homozygous, depending on whether a set of homologous chromosomes (pairs of choromosomes with one coming from the father, another from mother) differ or are identical on that particular position, respectively.

One important aspect of modern day genomics is identifying genes or SNPs responsible for certain diseases. Given SNPs from two groups of users–case (individuals showing traits of the disease) and control (individuals representing healthy population)–one can perform Pearson’s χ^2 test of association to determine whether presence of certain SNP is associated to disease susceptibility or not. SNPs with high χ^2 statistic are thought to be responsible for the disease.

3.1 Track 2 of the iDASH 2017 Challenge: χ^2 Test for Whole Genome Variants Search

The goal of Track 2 of the iDASH Privacy & Security Workshop 2017 competition [iDa17] was to develop a scalable and secure solution using SGX technology for whole genome variants search among multiple individuals. The input data is Variant Call Format (VCF) files containing sensitive SNP information from case and control groups of users. Logically, a single VCF file corresponds to a single individual and is a collection of SNPs, along with the information whether the SNPs are homozygous or heterozygous. Suppose we have n_1 case users and n_2 control users. To evaluate χ^2 statistic for a particular SNP s, one needs to find out how many times it is present among case and control users by single counting heterozygous occurrences and double counting homozygous occurrences. Suppose a_s is the count of SNP s among case users and a'_s among control users. Note, $(2n_1 - a_s)$ and $(2n_2 - a'_s)$ are the absence counts of SNP s among case and control users. Now, for the SNP s observed frequencies O_s, and expected frequencies (assuming no association) E_s can be stated as

$$
\begin{aligned}
O_s &= [a_s, a'_s, 2n_1 - a_s, 2n_2 - a'_s] \\
E_s &= [r(a_s + a'_s), (1-r)(a_s + a'_s), 2n_1 - r(a_s + a'_s), 2n_2 - (1-r)(a_s + a'_s)],
\end{aligned}
$$

where the ratio $r = \frac{n_1}{n_1+n_2}$. From the observed and expected frequencies the χ^2 test statistic for SNP s can be calculated as

$$\sum_{j=0}^{3} \frac{(O_s[j] - E_s[j])^2}{E_s[j]}. \tag{1}$$

The p-value for the SNP s is the probability that a random variable following a χ^2 distribution with degree of freedom one[3] attains a value larger than the computed test statistic. To find the top k most significant SNPs, one needs to compute p-values for all SNPs present in the genome data set and output k SNPs with least p-values or equivalently output k SNPs with highest χ^2 test statistic.

In the iDash competition pre-processing and compression of individual VCF files were allowed, with the constraint that any operation involving multiple VCF files cannot be performed at the pre-processing stage. It must be done inside the SGX enclave. This constraint correctly depicts the real life use case, where each VCF file is owned by the corresponding human individual. They can pre-process and compress their own information and send it to remote SGX enclave running on a possibly adversarial computational server. Honest individuals following the protocol are not expected to communicate among them, they are only supposed to send their information to the SGX enclave running in the computational server.

[3] χ^2 distribution with degree of freedom d is defined as sum of square of d independent standard normal variables.

The computationally expensive step in the above calculation is finding out 'count' of every SNP among case and control users. The natural way to evaluate these count values is as follows.

- represent individual VCF files as dictionaries (collection of (key, value) pairs) as follows:
  - For an user u belonging to the Case group, for all SNPs s present in its VCF file we have
$$\begin{aligned}\text{Dict}_u[s].\text{Case} &= \begin{cases}1, & \text{if } s \text{ is heterozygous for user } u\\ 2, & \text{if } s \text{ is homozygous for user } u\end{cases}\\ \text{Dict}_u[s].\text{Cont} &= 0\end{aligned}$$
  - For user v in the Control group it is exactly the opposite. That is for all SNP s present in user v's VCF file $\text{Dict}_v[s].\text{Case} = 0$ and $\text{Dict}_v[s].\text{Cont}$ is either one or two depending whether s is heterozygous or homozygous for user u.

  If we query the dictionary Dict_u with any SNP not present in user u's VCF file, it returns zero in both case and control counters. In other words $s' \notin \text{Dict}_u.\text{Keys}$, we have $\text{Dict}_u[s'].\text{Case} = \text{Dict}_u[s'].\text{Cont} = 0$.
- Merge all user dictionaries. Where the dictionary merging operation is defined as follows. For all $s \in \text{Dict}_A.\text{Keys} \cup \text{Dict}_B.\text{Keys}$,
$$(\text{Dict}_A \cup \text{Dict}_B)[s].\text{Case} = \text{Dict}_A[s].\text{Case} + \text{Dict}_B[s].\text{Case}$$
$$(\text{Dict}_A \cup \text{Dict}_B)[s].\text{Cont} = \text{Dict}_A[s].\text{Cont} + \text{Dict}_B[s].\text{Cont}$$

  After merging we have the merged dictionary
$$\text{Dict}_{\text{Merge}} = \cup_{u \in \text{case users} \cup \text{control users}} \text{Dict}_u$$

$\text{Dict}_{\text{Merge}}$ contains count of SNPs among case and control users. After building the dictionary rest of the calculation is relatively straight–forward. The whole process is described in Algorithm 3, where Merge is the functionality that takes dictionaries (containing SNPs as keys and corresponding counter as value) as input, and the merged dictionary as output. In other words
$$\text{Merge}(\text{Dict}_1, \cdots, \text{Dict}_n) \to \text{Dict}_1 \cup \cdots \cup, \text{Dict}_n.$$

`CalcChiSquare` is a function that takes
$$\big(\text{number of case users, number of control users}, (snp, (\text{Case}, \text{Cont}))\big)$$
as input and outputs $(snp, \chi^2\text{-statistic})$ where χ^2-statistic is calculated according to Equation (1). $\texttt{ForEach}^f(\mathcal{V})$ is a functionality which outputs the list $\{f(v) : v \in \mathcal{V}\}$.

Algorithm 3 Genome variants search to find top k SNPs

```
INPUT: : User set $\mathcal{U} = \mathcal{U}_{\text{Case}} \cup \mathcal{U}_{\text{Cont}}$ and SNP dictionaries $\text{Dict}_u$ for all $u \in \mathcal{U}$, size
    of case and control user groups.
OUTPUT: : Top $k$ SNPs $(snp_1, \cdots, snp_k)$
1: procedure GVS($\{\text{Dict}_u : \forall u \in \mathcal{U}\}, n_1 = \|\mathcal{U}_{\text{Case}}\|, n_2 = \|\mathcal{U}_{\text{Cont}}\|$)
2:     $\text{Dict}_{\text{Merge}} \leftarrow \text{Merge}(\{\text{Dict}_u : u \in \mathcal{U}\})$
3:     $\text{List}_{SNP} \leftarrow \texttt{ForEach}^{\textsc{CalcChiSquare}(n_1, n_2, \cdot)}(\text{Dict}_{\text{Merge}})$
4:     $\text{List}_{SNP}.\textsc{Sort}()$ ▷ Sorts the list in a decreasing order based on chisquare value
5:     return $\text{List}_{SNP}[1:k].snp$    ▷ Output top $k$ SNPs
6: end procedure
```

3.2 The Winning Solution of the iDash Track 2 Challenge [CT18]

The main challenge in the above computation is memory access optimization. The input data size is in the order of tens of gigabytes, whereas SGX enclaves are limited to about 96 MegaBytes of usable memory without paging. Moreover, inside SGX enclaves, the last level cache (LLC) miss penalty is considerably higher compared to native execution because this requires an extra round of encryption/decryption. This extra cost is by design, because in SGX architecture the main random access memory (RAM) always remains encrypted.

In Algorithm 3 the size of the $\text{Dict}_{\text{case}}$ and $\text{Dict}_{\text{cont}}$ dictionaries become the bottleneck. Even if we compress the SNPs and keep a single dictionary with separate case and control counters, we need at least 4 bytes to encode a SNP and $2 + 2 = 4$ bytes to store the two counters. However, in the sample data set provided in the competition there were about 5.5 Million unique SNPs. This means a trivial lower bound for the total size of the merged dictionaries is $(4 + 4) * 5.5 = 44$ MB. Even though, this lower bound is well short of the 96 MB limit to avoid page faults, this is far bigger than typical LLC size which is 6 or 8 MB. A typical memory efficient dictionary or hash-map implementation usually involves a random memory access for each key access. This leads to an almost mandatory cache fault for every dictionary access if we cannot fit the dictionary inside the LLC. As a result, any SGX implementation of the Section 3.1 algorithm typically incurs about a factor of two slowdown compared to native execution. To address this issue Carpov and Tortech [CT18] adopted an ingenious yet simple horizontal partitioning technique to reduce the memory requirement so that everything could be done within the LLC. The key observation is instead of building the large dictionary containing all SNPs and then finding the top k SNPs among them, we can partition the SNPs in various batches and process each batch independently while updating a global list of the top k SNPs.

We can divide the key space $\mathcal{K}$ (all possible values of SNPs) of the dictionaries into n disjoint parts $\mathcal{K}_1, \cdots, \mathcal{K}_n$. This in turn divides each user dictionary Dict_u into n disjoint smaller dictionaries $\text{Dict}_{u,1}, \cdots, \text{Dict}_{u,n}$ such that

$$\text{Dict}_u = \text{Dict}_{u,1} \cup \cdots \cup \text{Dict}_{u,n}.$$

Algorithm 4 Cache friendly Genome variants search to find top k SNPs

```
INPUT: : User set $\mathcal{U} = \mathcal{U}_{\text{Case}} \cup \mathcal{U}_{\text{Cont}}$ and SNP dictionaries $\text{Dict}_{u,i}$ for all $u \in \mathcal{U}$
    and $i \in [1, n]$, size of case and control user groups.
OUTPUT: : Top $k$ SNPs $(snp_1, \cdots, snp_k)$
 1: procedure GVS($\{\text{Dict}_{u,i} : \forall u \in \mathcal{U}, i \in [1, n]\}, n_1 = \|\mathcal{U}_{\text{Case}}\|, n_2 = \|\mathcal{U}_{\text{Cont}}\|$)
 2:     $\text{List}_{SNP} \leftarrow \Phi$
 3:     for all $i \in [1, n]$ do
 4:         $\text{Dict}_{\text{Merge},i} \leftarrow \text{Merge}(\{\text{Dict}_{u,i} : u \in \mathcal{U}\})$
 5:         $\text{List}_{Temp} \leftarrow \texttt{ForEach}^{\textsc{CalcChiSquare}(n_1, n_2, \cdot)}(\text{Dict}_{\text{Merge},i})$
 6:         $\text{List}_{SNP}.\textsc{Insert}(\text{List}_{Temp})$
 7:         $\text{List}_{SNP}.\textsc{Sort}()$   ▷ Sorts the list in a decreasing order based on chisquare
    value
 8:         $\text{List}_{SNP} \leftarrow \text{List}_{SNP}[1 : k]$          ▷ Only keep top $k$ elements of the list
 9:     end for
10:     return $\text{List}_{SNP}$                                  ▷ Output top $k$ SNPs
11: end procedure
```

4 Oblivious Genome Variants Search

Algorithms described in the previous section are not memory oblivious in general. In this section, we show under certain conditions the algorithms can be implemented in a memory oblivious way. Non memory oblivious SGX implementations might leak some non trivial information such as number of common SNPs among any two persons. Moreover, if some of the individuals are malicious and they collude with the server they can figure out exactly which SNPs are present in other individual's VCF files. $\mathcal{U}_{\text{Case}}$ be the set of users belonging to the case group and $\mathcal{U}_{\text{Cont}}$ be the set of users belonging to the control group. Every user $u \in \mathcal{U}_{\text{Case}} \cup \mathcal{U}_{\text{Cont}}$ sends their input I_u to a centralized server $\mathcal{S}$, which runs a Genome Variants Search algorithm inside its SGX enclave. It's worth mentioning that every user u must

- perform a remote attestation with S, to ensure it is running the appropriate executable inside SGX enclave and
- perform a key exchange with the enclave and send I_u by encrypting and authenticating with the exchanged key,

to ensure the data can only be accessed by the enclave. I_u is some encoding of the VCF data corresponding to user u. In Algorithm 3 we have $I_u = \text{Dict}_u$, where as in Algorithm 4 we have $I_u = \{\text{Dict}_{u,i} : i \in [1, n]\}$. GVS be the function which takes I_u's as input and outputs top k SNPs.

From a high level perspective the Genome Variant Search algorithms described in previous section have three distinct steps:

1. Merge input dictionaries to form a merged dictionary.
2. Calculate chi-Square statistic for each entry.
3. Sort the dictionary entries based on the chi-square statistic.

Chi-square statistic calculation is trivially memory oblivious (can be implemented by an arithmetic circuit). There are many well known perfectly memory oblivious sorting [AKS83,CGLS18,Bat68,Goo14] techniques which do not leak any side information. In Section 5 we discuss how to obliviously implement the dictionary Merge routine under various reasonable leakage functions. Once we have oblivious implementations of the dictionary merge routine and sort routine, next theorems show we can quantify the leakage in Algorithm 3 and Algorithm 4.

Theorem 1. *If the* Merge *routine in Algorithm 3 is implemented obliviously with leakage function* $\texttt{leakage}_{\text{Merge}}$ *and the* `Sort` *routine in line 3 is implemented by some perfect oblivious sort implementation, then Algorithm 3 becomes an oblivious implementation of GVS with leakage function* $\texttt{leakage}_{GVS}$*, where*

$$\begin{aligned}&\texttt{leakage}_{GVS}(\{\text{Dict}_u : \forall u \in \mathcal{U}\})\\ =&\texttt{leakage}_{\text{Merge}}(\{\text{Dict}_u : \forall u \in \mathcal{U}\}) \cup \{\|\text{Dict}_{\text{Merge}}\|, \|\mathcal{U}_{\text{Case}}\|, \|\mathcal{U}_{\text{Cont}}\|\}.\end{aligned}$$

Proof Sketch. We construct a simulator for GVS, given a simulator for Merge, given as Algorithm 5. We can show that the real algorithm is indistinguishable from the simulator by hopping over a single hybrid. In the hybrid, we replace the merging step with the corresponding simulator, which just takes the leakage due to the merge as input. In the next hop, which is to the final simulator, we sample Dict_u from random, instead of using the real input Dict_u. The sampling is done as follows: first $\|\text{Dict}_{\text{Merge}}\|$ number of unique keys are sampled from the domain of keys. Then the values of Dict_u at those keys are assigned arbitrarily. We recall that the in construction of the merged list the input array is scanned linearly and the number of positions scanned only depends on the number of entries, *i.e.*, $\|\text{Dict}_u\|$, and not their values. Hence the addresses utilized in this part of the simulator would be indistinguishable from the hybrid.

Algorithm 5 Simulator for GVS

```
INPUT: leakage_Merge({Dict_u : ∀u ∈ U}) ∪ (‖Dict_Merge‖, n_1 = ‖U_Case‖, n_2 = ‖U_Cont‖).
OUTPUT: addresses.
1: procedure SIM-GVS($\texttt{leakage}_{\text{Merge}}(\{\text{Dict}_u : \forall u \in \mathcal{U}\}) \cup (\|\text{Dict}_{\text{Merge}}\|, n_1, n_2)$)
2:     $\text{List}_{SNP} \leftarrow \emptyset$.
3:     addresses-dict-merge ← SIM-MERGE($\texttt{leakage}_{\text{Merge}}(\{\text{Dict}_u : \forall u \in \mathcal{U}\})$)
4:     Sample $\text{Dict}_{\text{Merge}}$ randomly, constrained by $\|\text{Dict}_{\text{Merge}}\|$.
5:     $\text{List}_{SNP} \leftarrow \texttt{ForEach}^{\text{CalcChiSquare}(n_1, n_2, \cdot)}(\text{Dict}_{\text{Merge}})$
6:     $\text{List}_{SNP}$.Sort()    ▷ Sorts the list in a decreasing order based on chisquare value
7:     addresses-extra ← Addresses used in Steps 5-6.
8:     return addresses-dict-merge, addresses-extra    ▷ Output all addresses
9: end procedure
```

Theorem 2. *If the* Merge *routine in Algorithm 4 is implemented obliviously with leakage function* $\mathtt{leakage}_{\text{Merge}}$ *and the* `Sort` *routine in line 6 is implemented by some perfect oblivious sort implementation, then Algorithm 4 becomes an oblivious implementation of GVS with leakage function* $\mathtt{leakage}_{GVS}$*, where*

$$\mathtt{leakage}_{GVS}(\{\text{Dict}_{u,i} : \forall u \in \mathcal{U}, i \in [1, n]\},)$$
$$= \Big(\mathtt{leakage}_{\text{Merge}}(\{\text{Dict}_{u,1} : \forall u \in \mathcal{U}\}), \cdots, \mathtt{leakage}_{\text{Merge}}(\{\text{Dict}_{u,n} : \forall u \in \mathcal{U}\})\Big)$$
$$\cup \{\|\text{Dict}_{\text{Merge},1}\|, \cdots, \|\text{Dict}_{\text{Merge},n}\|, \|\mathcal{U}_{\text{Case}}\|, \|\mathcal{U}_{\text{Cont}}\|\}$$

The proof of this theorem is fairly similar to the last one: instead of simulating the merge monolithically, the simulation is done partition by partition. The arguments for the rest of the algorithm carry over straightforwardly.

5 Oblivious Dictionary Merging

In the previous section, we showed that given a procedure to obliviously merge multiple dictionaries we can obliviously implement the Genome Variants Search algorithms. In this section show how oblivious dictionary merging can be done.

In Section 3.1, we defined the notion of dictionary merging in the context of genome variants search. However, the algorithms described in this section work for generic dictionary merging operations. A dictionary or associative array Dict is a dynamic collection of $(\mathtt{key}, \mathtt{value})$ pairs, such that each possible *key* appears only once in the collection. It usually supports `insert`, `delete`, `update` and `lookup` operations based on the `key`. The operator [ ] is used as an access operator. That is if $(\mathtt{key}, \mathtt{value}) \in$ Dict, then Dict[`key`] returns a reference to `value`. Let Dict.Keys denote the set of all `key`s in the dictionary. For any $\mathtt{key} \notin$ Dict.Keys,

- as rvalue Dict[`key`] returns `Null`. In other words $\mathtt{value} = \text{Dict}[\mathtt{key}]$ sets the variable `value` to `Null`.
- as lvalue Dict[`key`] inserts a pair $(\mathtt{key}, \mathtt{value})$ to the dictionary and returns a reference to the variable `value`. In other words $\text{Dict}[\mathtt{key}] = \mathtt{value}$ inserts $(\mathtt{key}, \mathtt{value})$ into the dictionary Dict.

Let $\mathcal{V}$ be the set of all possible `value`s excluding `Null`. $\oplus$ be a binary operator over $\mathcal{V}$. It can be naturally extended to $\mathcal{V} \cup \{\mathtt{Null}\}$ as follows. For any $\mathtt{value} \in \mathcal{V}$,

$$\mathtt{value} \oplus \mathtt{Null} = \mathtt{value}, \qquad \mathtt{Null} \oplus \mathtt{value} = \mathtt{value}, \qquad \mathtt{Null} \oplus \mathtt{Null} = \mathtt{Null}.$$

For the Genome Variants Search application described in Section 3.1, $\oplus$ operator over (Case, Cont) pairs is defined as $a \oplus b = (a.\text{Case} + b.\text{Case}, a.\text{Cont} + b.\text{Cont})$. For any two dictionaries Dict_1 and Dict_2, the Merge operation (also represented by the operator $\cup$) is defined as follows. First $(\text{Dict}_1 \cup \text{Dict}_2).\text{Keys} = \text{Dict}_1.\text{Keys} \cup \text{Dict}_2.\text{Keys}$. Second for all $\mathtt{key} \in (\text{Dict}_1 \cup \text{Dict}_2).\text{Keys}$, we have $(\text{Dict}_1 \cup \text{Dict}_2)[key] = \text{Dict}_1[key] \oplus \text{Dict}_2[key]$.

For more than two dictionaries the Merge operation is defined inductively. For $n \geq 2$, we have

$$\mathrm{Dict}_1 \cup \cdots \cup \mathrm{Dict}_n = (\mathrm{Dict}_1 \cup \cdots \cup \mathrm{Dict}_{n-1}) \cup \mathrm{Dict}_n.$$

Dictionaries or hash tables are usually implemented either by chaining or by open addressing. [Che17] is a short summary and comparison of various hash table implementations. It suggests open address based hash table implementation Robin Hood[CLM85] is probably the fastest memory efficient implementation. For the purpose of this paper, we will assume the hash table memory is contiguous, which is the case for all open addressing based hash tables. This means we can sequentially access all elements of the hash table by a linear sweep. The memory addresses accessed in this operation are independent of the hash table content.

The location of any ($\texttt{key}, \texttt{value}$) pair in the contiguous memory is determined by $\texttt{hash}(\texttt{key})$. We will assume the function $\texttt{hash}$ is a random oracle [BR93], to ensure that the pair ($\texttt{key}, \texttt{value}$) gets stored in a random location independent of the variable $\texttt{key}$. The idea behind an almost ideal (in terms of leakage) dictionary merging is pretty simple and can be described in three high level steps:

1. Sequentially access all ($\texttt{key}, \texttt{value}$) pairs of all input dictionaries $\mathrm{Dict}_1, \cdots, \mathrm{Dict}_n$ and store them in a large array $\texttt{Array}$ of size $\|\mathrm{Dict}_1\| + \cdots + \|\mathrm{Dict}_n\|$.
2. Obliviously shuffle $\texttt{Array}$ and generate $\texttt{Array}'$.
3. Build the new dictionary $\mathrm{Dict}_{\mathrm{Merge}}$ by sequentially traversing $\texttt{Array}'$.

The memory access pattern in first two steps are completely independent of the input data. However, the last step leaks some non trivial information. A resourceful adversary can track how the memory locations within the contiguous storage are being accessed. The location of a dictionary entry corresponding to $\texttt{key}$ gets determined by $\texttt{hash}(\texttt{key})$. By the random oracle property of the $\texttt{hash}$ function, the location does not reveal anything about the content of $\texttt{key}$. Also, because of the oblivious shuffle this address does not reveal from which input dictionary Dict_i the $\texttt{key}$ is coming from. But the adversary can observe how many times each address location is getting accessed. This in turn leaks the collision distribution of the input dictionaries, which is essentially the following information.

$$\sum_{i=1}^{n} \|\mathrm{Dict}_i\|, \sum_{1 \leq i < j \leq n} \|\mathrm{Dict}_i \cap \mathrm{Dict}_j\|, \quad \sum_{1 \leq i < j < k \leq n} \|\mathrm{Dict}_i \cap \mathrm{Dict}_j \cap \mathrm{Dict}_k\|,$$
$$\cdots, \|\mathrm{Dict}_1 \cap \mathrm{Dict}_2 \cap \cdots \cap \mathrm{Dict}_n\|.$$

6 Experimental Results

For our experimental results, we use the public dataset available as part of the iDash 2017 competition [iDa17]. The dataset consists of VCF files from two groups of individuals, case group Case whose members show symptoms of some particular disease and control group Cont consisting of healthy individuals. The total size of the two thousand VCF files is about 27.4 GB. We ran our experiments on an Intel NUC6i7KYK, which has 6 MB of LLC. In comparison, the platform used in [CT18] had 8 MB LLC size. Our baseline implementation takes 28 seconds for pre-processing (or total time for client side computation). In the baseline non oblivious implementation of Algorithm 4, the computation time inside the SGX enclave is 16 seconds. On the other hand, the winning candidate from [CT18] reports about 65 seconds of pre-processing time and 7 seconds of enclave runing time. The pre-processing is mainly bounded by the SSD read write speed. Our pre-processing is faster because we used a larger block size (every VCF file is divided only in 15 parts). On the other hand, [CT18]'s enclave running time is almost half that of ours for two main reasons: first, our enclave is single threaded, as opposed to 8 threads in [CT18]. In fact, our dictionary implementation is not thread safe. Second, we have only 6 MB of cache memory. [CT18] had 8 MB.

The oblivious dictionary merging algorithm described in Section 5 has a crucial drawback. It requires oblivious shuffling of a very large array containing all the input data. After compression the total size of input data is about 4.5 GB. In the baseline implementation we partitioned the data in 15 parts, to fit individual dictionaries inside the LLC. After this partitioning, the output of each Merge call in Algorithm 4 fits well within the LLC, but the input is still large: about 4.5GB /15 = 300 MB. For an efficient memory oblivious shuffle we needed to fit the input data within LLC. To address this we can further partition the input data and shuffle each partition independently. This partitioning actually leaks more information, such as the collision patterns among different partitions. In our implementation in every partition we take 256 SNPs each from 16 users and shuffle them together. We used Batcher's bitonic merge sort algorithm [Bat68] for oblivious shuffling. We also used SipHash [AB12] as our choice of random oracle. To our knowledge, this is the fastest known pseudorandom function for short input sizes. In this parameter setting the enclave running time is about 5 minutes. This shows even though memory oblivious implementation is practical if we are willing to leak some amount of collision distributions, it is still considerably slower than the non oblivious implementation. One thing to note is that the performance of the scheme is dependent upon the choice of data partitioning and hence information leakage. Finding a better data partitioning technique which would allow minimal leakage and the fastest possible performance remains an open problem.

References

[AB12] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A fast short-input PRF. In Steven D. Galbraith and Mridul Nandi, editors, *Progress in Cryptology - INDOCRYPT 2012: 13th International Conference in Cryptology in India*, volume 7668 of *Lecture Notes in Computer Science*, pages

489–508, Kolkata, India, December 9–12, 2012. Springer, Heidelberg, Germany.

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *15th Annual ACM Symposium on Theory of Computing*, pages 1–9, Boston, MA, USA, April 25–27, 1983. ACM Press.

[ATG$^+$16] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, et al. Scone: Secure linux containers with intel sgx. In *OSDI*, volume 16, pages 689–703, 2016.

[Bat68] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314. ACM, 1968.

[BMD$^+$17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: Sgx cache attacks are practical. *arXiv preprint arXiv:1702.07521*, page 33, 2017.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93: 1st Conference on Computer and Communications Security*, pages 62–73, Fairfax, Virginia, USA, November 3–5, 1993. ACM Press.

[CGLS18] TH Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Cache-oblivious and data-oblivious sorting and applications. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2201–2220. SIAM, 2018.

[Che17] Felix Chern. Writing a damn fast hash table with tiny memory footprints. `http://www.idryman.org/blog/2017/05/03/writing-a-damn-fast-hash-table-with-tiny-memory-footprints`, 2017. Accessed: 2018, June 7.

[CLM85] Pedro Celis, Per-Åke Larson, and J. Ian Munro. Robin Hood hashing (preliminary report). In *26th Annual Symposium on Foundations of Computer Science*, pages 281–288, Portland, Oregon, October 21–23, 1985. IEEE Computer Society Press.

[CT18] Sergiu Carpov and Thibaud Tortech. Secure top most significant genome variants search: idash 2017 competition. Cryptology ePrint Archive, Report 2018/314, 2018. `https://eprint.iacr.org/2018/314`.

[EZ17] Saba Eskandarian and Matei Zaharia. An oblivious general-purpose sql database for the cloud. *arXiv preprint arXiv:1710.00458*, 2017.

[FVBG17] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. IRON: Functional encryption using intel SGX. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17: 24th Conference on Computer and Communications Security*, pages 765–782, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[GESM17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, pages 2:1–2:6, New York, NY, USA, 2017. ACM.

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 182–194, New York City, NY, USA, May 25–27, 1987. ACM Press.

[Goo14] Michael T. Goodrich. Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In David B. Shmoys, editor, *46th Annual ACM Symposium on Theory of Computing*, pages 684–693, New York, NY, USA, May 31 – June 3, 2014. ACM Press.

[iDa17] Idash privacy & security workshop. `http://www.humangenomeprivacy.org/2017/competition-tasks.html`, 2017. Accessed: 2018, June 7.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113, Santa Barbara, CA, USA, August 18–22, 1996. Springer, Heidelberg, Germany.

[KSWH98] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In Jean-Jacques Quisquater, Yves Deswarte, Catherine Meadows, and Dieter Gollmann, editors, *ESORICS'98: 5th European Symposium on Research in Computer Security*, volume 1485 of *Lecture Notes in Computer Science*, pages 97–110, Louvain-la-Neuve, Belgium, September 16–18, 1998. Springer, Heidelberg, Germany.

[MDS99] Thomas S Messerges, Ezzy A Dabbish, and Robert H Sloan. Investigations of power analysis attacks on smartcards. *Smartcard*, 99:151–161, 1999.

[NFR$^+$17] Kartik Nayak, Christopher W. Fletcher, Ling Ren, Nishanth Chandran, Satya V. Lokam, Elaine Shi, and Vipul Goyal. HOP: Hardware makes obfuscation practical. In *ISOC Network and Distributed System Security Symposium – NDSS 2017*, San Diego, CA, USA, 2017. The Internet Society.

[OGTU14] Olga Ohrimenko, Michael T. Goodrich, Roberto Tamassia, and Eli Upfal. The melbourne shuffle: Improving oblivious storage in the cloud. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *ICALP 2014: 41st International Colloquium on Automata, Languages and Programming, Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 556–567, Copenhagen, Denmark, July 8–11, 2014. Springer, Heidelberg, Germany.

[OSF$^+$16] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium*, pages 619–636, 2016.

[Pag02] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive, Report 2002/169, 2002. `http://eprint.iacr.org/2002/169`.

[SCF$^+$15] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press.

[SCNS16] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang, editors, *ASIACCS 16: 11th ACM Symposium on Information, Computer and Communications Security*, pages 317–328, Xi'an, China, May 30 – June 3, 2016. ACM Press.

[SLK$^+$17] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-shield: Enabling address space layout randomization for SGX programs. In *ISOC Network and Distributed System Security Symposium – NDSS 2017*, San Diego, CA, USA, 2017. The Internet Society.

[SLKP17] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *ISOC Network and Distributed System Security Symposium – NDSS 2017*, San Diego, CA, USA, 2017. The Internet Society.

[SVDS$^+$13a] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

[SvDS$^+$13b] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 299–310, Berlin, Germany, November 4–8, 2013. ACM Press.

[VFP$^+$15] Sabrina De Capitani Di Vimercati, Sara Foresti, Stefano Paraboschi, Gerardo Pelosi, and Pierangela Samarati. Shuffle index: efficient and private access to outsourced data. *ACM Transactions on Storage (TOS)*, 11(4):19, 2015.

[WKPK16] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting synchronisation bugs in intel SGX enclaves. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *ESORICS 2016: 21st European Symposium on Research in Computer Security, Part I*, volume 9878 of *Lecture Notes in Computer Science*, pages 440–457, Heraklion, Greece, September 26–30, 2016. Springer, Heidelberg, Germany.

[ZCC$^+$16] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16: 23rd Conference on Computer and Communications Security*, pages 270–282, Vienna, Austria, October 24–28, 2016. ACM Press.

[ZDB$^+$17] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, pages 283–298, 2017.