Parallel Oblivious Sorting in SGX Enclave

Tianyao Gu, Tian Xie

December 2023

Abstract

Oblivious algorithms have been a popular research topic in cybersecurity as it guarantees a simulatable memory access pattern and thereby provably resists side channel attacks. We implemented a parallel oblivious sorting algorithm in C++ using Intel SGX Enclave, a hardware-based trusted-computing environment offering privacy and authenticity. We leveraged OpenMP to achieve multithreading on a 36-core CPU, and further improves parallelism through SIMD. We increased the arithmetic density of the algorithm to make it efficient in an external memory model. We optimized the memory utilization of the program to minimize the consumption of Enclave Page Cache. We also overlapped computation and communication through prefetching and write back buffers.

1 Background

1.1 Background on Oblivious Sorting Algorithms

Our motivation is to securely outsource data and computation to an untrusted worker equipped with secure processors such as Intel SGX. Although secure processors ensure data privacy through encryption, existing literature reveals that attackers can exploit memory access and page swap patterns during computations to glean information about the data.

Oblivious algorithms provide a verifiable shield to counter such side-channel attacks. This is because "Obliviousness" essentially demands that memory access and page swap patterns remain independent of secret data.

In addition, sorting stands out as a fundamental building block of various oblivious computation applications. Specifically, oblivious sorting is key to common graph algorithms [2, 3, 5] including breadth-first search [2, 3], connected components [6], minimum spanning tree/forest [6], clustering [5], list ranking [6], tree computations with Euler tour [6], and tree contraction [6]. Oblivious sorting can also be used for securely initializing common ORAM algorithms [9] including Path ORAM [7], which has been deployed at a large scale in practice [8]. Moreover, any computational task that can be efficiently expressed as a streaming-Map-Reduce algorithm has an efficient oblivious implementation using oblivious sort [3, 5].

Our project is based on a prior research paper by Gu et al. [4]. The paper proposed and implemented an efficient oblivious sorting in hardware enclaves, which achieves asymptotically optimal runtime and outperforms all the previous works concretely. However, the algorithm, as presented and implemented, is single-threaded, posing limitations on its performance.

In our continuation of this work, we aim to enhance performance by parallelizing the oblivious sorting algorithm. As a byproduct, we obtain a parallel oblivious random shuffling algorithm, which is also a crucial primitive in oblivious computation.

1.2 Background on Flex-way Butterfly Oblivious Sort

To achieve oblivious sorting, Asharov et al. [1] proposed the idea to first design an *oblivious random bucket assignment* algorithm, which randomly assigns each input element to one of $O(N/Z)$ output buckets, each of poly-logarithmic capacity Z; further, the obliviousness property requires that the access patterns do not leak the destination bucket of each element. Then, from the oblivious random bucket assignment, they get an *oblivious shuffling* algorithm, which randomly permutes the inputs without revealing the permutation. Finally, they can apply any non-oblivious, comparison-based sorting algorithm to the shuffled array.

Flex-way Butterfly Oblivious Sort [4] adopts this idea and achieved the *oblivious random bucket assignment* using a multi-way butterfly network. The input of the network consists of $(1+\epsilon)N/Z$ buckets of size Z, where each bucket contains $Z/(1+\epsilon)$ real input elements and is padded with filler elements. Each element is attached a random tag.

The network relies on an important building block called multi-way MergeSplit, which merges elements from k buckets and obliviously splits the real elements into k buckets according to the modulus of their tags. Because each bucket is padded with filler elements, the no bucket will overflow except with negligible probability. Figure 1 demonstrates a 3-level butterfly network that assigns elements to 18 buckets.

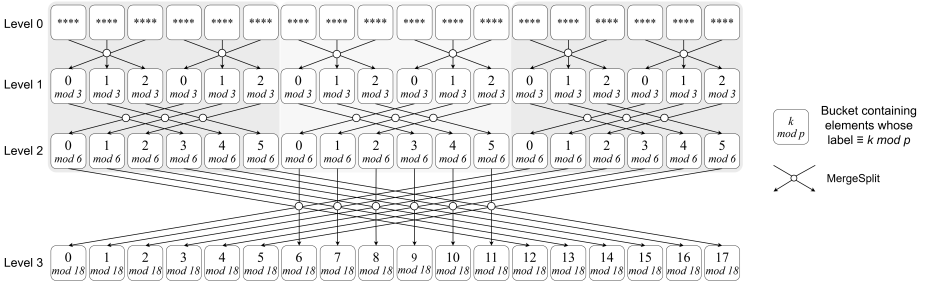

Figure 1: A 3-level multi-way butterfly network.

When implementing the algorithm in Intel SGX, we need to consider the

memory limitations of the SGX enclave. When transferring data across the boundary of the enclave, we must pay an expensive cost for encryption, decryption, and authentication. Naively emulating the butterfly network level by level would incur an extra I/O overhead of $\Theta(\log N)$. Therefore, we want to route elements through multiple levels each time we fetch them into the enclave, as is demonstrated in Figure 2. This optimization significantly increases the arithmetic density of the algorithm and thereby improves performance.

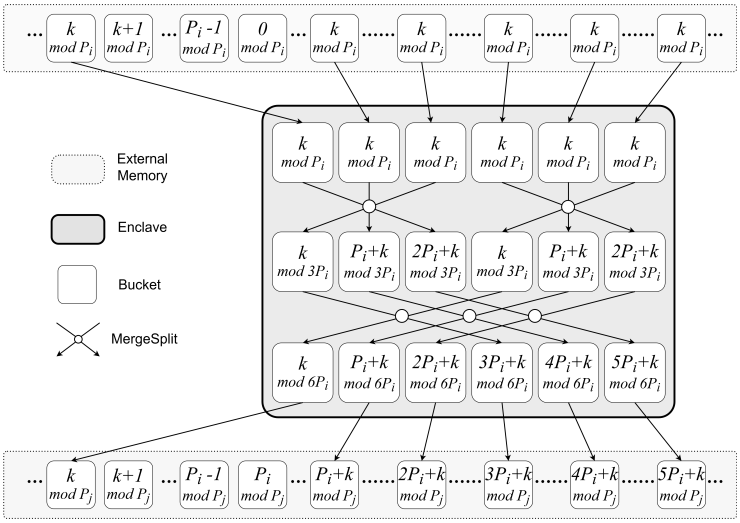

Figure 2: Route elements through multiple levels of butterfly network in each batch.

2 Approach

2.1 Identify Parallelism

3 Results