OGAPI : Oblivious Graph Processing in Multicores

Masab Ahmad, Omer Khan
University of Connecticut, Storrs, CT, USA
{masab.ahmad, khan}@uconn.edu

***Abstract*—Graph processing has become important problem for ultra-efficient embedded computing. However, malicious entities snooping on graph access/information leakage patterns might violate privacy. Examples of such exploitations include leakage of important graph vertices and other crucial algorithmic phases that can lead to privacy setbacks in various applications. Prior works focus on hardware schemes to mitigate associated memory leakage patterns, which come with dreadful overheads in the context of irregular graph workloads. In this work, we present algorithm level mechanisms to minimize or completely eliminate graph access leakage across all possible channels. We employ mechanisms centered around algorithmic redundancy to hide graph access patterns, and we show how effective parallelization strategy can allow performance improvements across these graph workloads. Our approach shows performance overheads of 2% over native executions of the shortest path graph workload.**

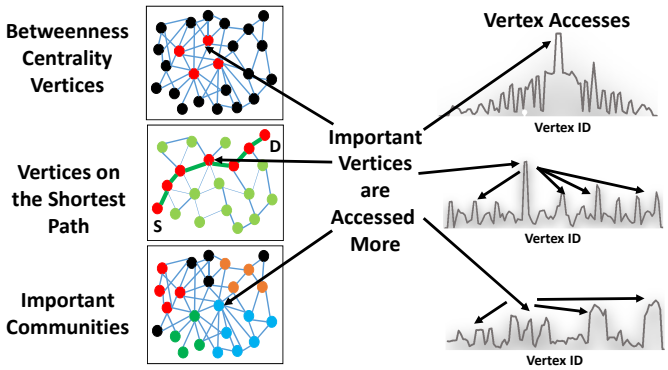

Fig. 1. How important vertices are accessed more in graph algorithms.

I. Introduction

Graph algorithms are highly ubiquitous in today's world [8]. With the advent of many-core processors such as multicores, exploitable parallelism is allowing these algorithms be deployed many embedded applications. Most algorithms scale to high thread counts, however exhibit weak scalability depending on how the data is accessed in these algorithms [1]. Graph algorithms, such as the ones that compute shortest paths, access data related to certain graph vertices more than other corresponding graph vertices [2]. This variation in accesses constructs data access patterns within memory subsystem, where certain vertices are accessed more than others.

Data access in graphs can be specified as vertex access, which pertains to graph access for a given algorithm. Patterns in data accesses are already known to be exploitable in prior works, with the idea being that an algorithm leaks more information when working on crucial data. Similarly, in the case of graph algorithms, vertex access patterns leak information about important vertices [3], as shown in Figure 1. Taking the example of an autonomous unmanned aerial drone that has path planning as a primary objective, graph access patterns can leak information pertaining to shortest paths. Malicious entities snooping around on the hardware computing these applications can infer where these drones are progressing, and can thus make them prone to sabotage. With many graph frameworks, such as Galois [8], mechanisms are thus required that reduce information leakage from the execution of such algorithms.

Prior work on trusted execution environments, such as Intel's software guard extensions (SGX) platform [6], has generally been completely hardware oriented, using ISA extensions to isolate program execution. However, these do not address the problem of side channels, that can occur anywhere in the architecture from interference on shared hardware channels [4]. A number of prior works apply oblivious data access schemes to algorithms using redundant data accesses, and prove that they do indeed eliminate leakage [9]. Using program redundancy, vertex access patterns can be made constant, so a malicious entity viewing channel leakage cannot distinguish between vertices, and thus cannot acquire any meaningful information [3]. However these works only show complexity overheads for sequential versions, and do not show any performance analysis and implications. Therefore, no prior work analyzes graph workloads in the context of vertex leakage in a parallel multicore setting, a setting where graph algorithms are ubiquitously applied today [8] [1].

Performance overheads take various overturns in multicores and other parallel paradigms, where parallelization strategy induces unpredictable synchronization and memory/data access overheads. However, they do present an opportunity to hide performance overheads associated with redundant data accesses in oblivious program execution. We propose our analysis and leakage elimination scheme in a parallel setting, and show that a software approach is much simpler with minimal performance overheads. Moreover, our approach hides the redundant work using an efficient parallelization strategy.

II. Quantifying Privacy vs. Efficiency

Shared hardware resources leak information in the form of bits to a snooping adversary. The theory behind this leakage is explained in [4], where authors use randomization and redundancy over time to bound bit leakage. We take an example of the shortest path workload (SSSP), and show how adding redundant work removes information leakage from vertex/data accesses. With an insecure baseline, only those

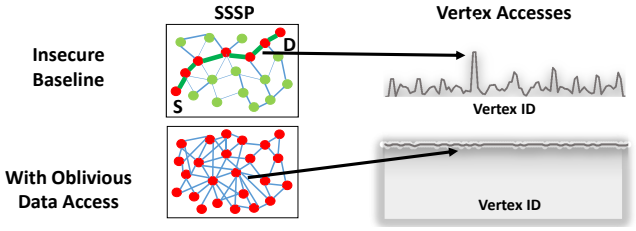

Fig. 2. Eliminating vertex leakage via algorithmic redundancy.

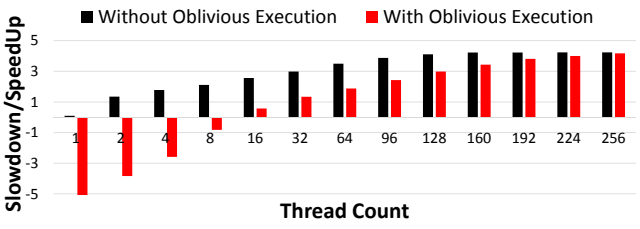

Fig. 3. Eliminating vertex leakage via algorithmic redundancy

vertices are accessed whose distances are minimal, and this results in more writes to cache lines where the requested data is located. Hence the vertex accesses to such vertices can be differentiated from other vertices, as shown in Fig 2. In the oblivious case, we add a uniform number of writes for each vertex computations, resulting in a combination of the actual algorithmic work as well as dummy computations. This causes adversaries to see the same number of accesses for all vertex computations, and thus it cannot infer any private information.

Once a user has chosen how much redundancy is required, one can efficiently parallelize the oblivious graph algorithm to hide its associated latency overheads. In a parallel scheme each thread performs some work on a chunk of vertices. With leakage control, each thread is expected not to leak any number of bits over time to an adversary across all possible channels. With extra work mainly in the form of compute and memory access, parallelism can efficiently hide associated additional latencies by simply dividing the redundant work amongst threads. However, the way redundant work is performed is highly application dependent, and thus communication between threads might become worse with parallelization. This offsets computation versus communication ratios in a multicore setting, which stops scalability of applications, degrading overall performance. These effects need to be studied in detail in practical settings in order to properly quantify tradeoffs between leakage and efficiency.

III. Methods and Results

One of the popular graph algorithm falls in the domain of finding a single source shortest path (SSSP). We take the SSSP workload from the CRONO suite [1], which contains several state-of-the-art parallel workloads interfaced with both real and synthetic graphs. The input graph used in this paper is the California (CA) road network graph [5]. The SSSP workload is executed on a simulated 256-core multicore using the Graphite Simulator [7]. Each core is modeled as an in-order pipeline with $32KB$ private L1 instruction and cache caches, and a $256KB$ shared L2 cache. The 256-core processor also models 8 memory controllers to access the off-chip memory. All input graphs for SSSP problem have an adjacency list representation.

Fig 3 shows overall speedups/slowdown obtained for the SSSP parallel workload relative to its sequential version. The x-axis shows the number of threads used to compute SSSP with and without oblivious (redundant) execution. For the CA road network, the average degree of the graph is around 2.5, while the maximum degree is around 12. So each vertex in the oblivious execution has to relax 12 edges to hide its data access pattern. The baseline SSSP algorithm's speedup obtained at 256 threads is $4.24\times$, which is in conjunction with [1]. This weak scalability is attributed to synchronization overhead, which is the major completion time component at high thread count [1]. However, with oblivious execution the overall speedup drops to $4.17\times$, which is a negligible performance overhead (around 2%). The primary reason for this is that the addition of redundant work is easily parallelized across threads, and thus the side effects of synchronization are less compared to the baseline scenario.

IV. Conclusion

In this paper we show how side channel information leakage can be minimized using software redundancy in state-of-the-art graph algorithms. In addition, we effectively use parallelization of the workload to hide the overheads of the redundant work. Our analysis of the shortest path workload shows an overhead of 2% at high thread count. As future work we plan to extend the idea of algorithmic redundancy and efficient parallelization to reduce the performance overheads of oblivious graph algorithms.

References

[1] M. Ahmad and et. al., "Crono : A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *Proceedings of the 2015 Annual IEEE Int. Symposium on Workload Characterization*, ser. IISWC. Washington, DC, USA: IEEE, 2015.

[2] M. Ahmad, K. Lakshminarasimhan, and O. Khan, "Efficient parallelization of path planning workload on single-chip shared-memory multicores," in *Proc. of the IEEE High Performance Extreme Computing Conf.*, ser. HPEC '15. IEEE, 2015.

[3] M. Blanton and et. al., "Data-oblivious graph algorithms for secure computation and outsourcing," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '13. NY, USA: ACM, 2013, pp. 207–218.

[4] C. Fletcher and et. al., "Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb 2014, pp. 213–224.

[5] J. Leskovec and et. al., "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," 2008.

[6] F. McKeen and et. al., "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd Int. Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. NY, USA: ACM, 2013, pp. 10:1–10:1.

[7] J. Miller and et. al., "Graphite: A distributed parallel simulator for multicores," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th Int. Symposium on*, Jan 2010, pp. 1–12.

[8] D. Nguyen and et. al., "Deterministic galois: On-demand, portable and parameterless," in *Proceedings of the 19th Int. Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. NY, USA: ACM, 2014, pp. 499–512.

[9] X. S. Wang and et. al., "Oblivious data structures," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. NY, USA: ACM, 2014, pp. 215–226.