Trace Oblivious Computation

Chang Liu
University of Maryland, USA
liuchang@cs.umd.edu

Abstract

In recent years, execution trace obliviousness has become an important security property in various applications in the presence of side channels. On the one hand, a cryptographic protocol called *Oblivious RAM* (ORAM) has been developed as a generic tool to achieve obliviousness, while incurring an overhead. On the other hand, customized oblivious algorithms with better performance have been developed. This method, however, is not scalable in terms of human efforts. This thesis work adopts a language design approach to facilitate users to develop efficient oblivious applications. I will study sequential and parallel programs and different channels, design languages and security type systems to support efficient algorithm implementations, while formally enforcing obliviousness. My study on the secure computation application shows that using our compiler, one PhD student can develop an oblivious algorithm in one day which took a research group of 5 researchers 4 months to develop in 2013, while achieving 10× to 20× better performance.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General

Keywords Oblivious Computation, Secure Type System

1. Motivation

In recent years, a trending paradigm of data and computation outsourcing has arisen in a wide range of applications ranging from privacy-preserving cloud computing, private outsourcing storage, to secure computation. In these applications, users allow their delegates to run their code over their sensitive data, and thereby relinquish control over both their intellectual property and their private information.

One important issue is how to protect users' data privacy in these applications. A straightforward idea is to encrypt their data. Though encryption is necessary, it is not sufficient in the presence of adversary who can observe a program's *execution trace*. For example, let us consider a typical cloud computing scenario where users transfer both their programs and their data to the provider. Prior work [12] has demonstrated that even if all data are encrypted, *memory addresses* transferred over the memory bus in plaintext allow attackers with physical access (e.g., a malicious employee of the cloud provider) to gain sensitive information.

To mitigate these threats, a security notion, called *obliviousness*, has been proposed. Intuitively, obliviousness requires that a program's execution trace patterns do not leak any information about sensitive data. For example, *Oblivious RAM* (ORAM) [2] can obfuscate memory addresses and protects memory access patterns. In particular, we can place sensitive code and data into ORAM, which has the effect of hiding the memory access pattern.

Prior work have demonstrated that generic ORAM simulation based approaches can achieve obliviousness for a class of traces (e.g. memory traces [5, 7], and instruction traces [6, 8]) at the expense of moderate ORAM overhead in practice. On the other hand, customized oblivious algorithms have been suggested for specific problems, to achieve asymptotically better overhead [9]. This approach, however, does not scale in terms of human effort.

My research focuses on developing novel programming techniques to enable practical oblivious computation. The idea is to develop tools to (1) reason whether a customized program is oblivious; (2) translate a program into its oblivious counterpart; and (3) achieve efficiency.

Challenges arise from both theoretical and practical sides. From a theoretical perspective, a general theory must be developed to provide principled methods to analyze programs' obliviousness. From a practical perspective, programming tools need be supplied to facilitate programmers to implement trace-oblivious applications. My dissertation will provide solutions addressing both concerns.

2. Problem

A general framework for trace obliviousness analysis. The first problem is how to formally enforce a program's obliviousness. My prior work proposed security type systems to enforce a program to be memory trace oblivi-

SPLASH Companion'15, October 25–30, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3722-9/15/10...$15.00
http://dx.doi.org/10.1145/2814189.2814200

ous [5, 7], and instruction trace oblivious [6, 8]. It remains unclear how to generalize these methods to model more side channels and to support richer program construction. First, my prior work most focused on leakage from the memory trace channel, but it is also possible to extend the work to protect timing channels, termination channel, program counter channel, etc. A generalized approach can provide a stronger security guarantee with respect to these additional side channels. Second, we currently consider deterministic programs, and many crytogrphic algorithms use random numbers and random permutations. I propose to extend the language with nondeterministic primitives to support these algorithms, including the ORAM algorithm itself.

Reasoning obliviousness for distributed programs. Most existing research focused on obliviousness of a sequential program. The second research problem is to study how to enforce obliviousness for distributed programs whose computation logic can be expressed in distributed languages such as Spark. Prior work [9] provides a specific oblivious solution for a class of parallel graph algorithms, but it lacks the generality to apply to customized algorithms that cannot be encoded as a graph algorithm (see Section 3 for more details). I will investigate language design and implementation choices allowing obliviousness analysis of a general class of parallel programs.

3. Approach

3.1 Generalizing Trace Obliviousness Analysis

I adopt a theory-practice combined approach to solve this problem. My existing work focused on two application scenarios that require obliviousness: privacy-preserving cloud computing and RAM-model secure computation.

Privacy-preserving cloud computing allows users outsource their data and computation to a cloud while retaining data's privacy. Particularly, we consider attackers with physical access, and the cloud providers are equipped with ORAM-capable secure processors. Using these secure processors, a generic approach prevents the attackers from learning any secrets by allocating all the data and code within one ORAM. We identified, however, that such a method is not always necessary, i.e., when data access pattern does not leak information.

Based on this observation, I designed a *compiler* which can allocate data intelligently into normal DRAM, encrypted memory, and ORAM banks, a *security type system* which can check if a program is memory trace oblivious with respect to a data allocation, and an *optimizer* that can optimize the code while enforcing the emitted target code is well typed. The challenges are (1) how to define the security notion, and show that the type system enforces security; and (2) how to type check assembly code in ORAM-capable ISA. I tackled these two problems in [5] and [7] respectively.

Secure computation allows two (or more) parties to jointly compute a public function over their secret data, while revealing no information except the output of the function. The basic idea is to express the computation in a circuit format, which can then be encoded using Yao's Garbed Circuits [3], and Garbed Circuits encrypts the computation output automatically. Since programmers favor RAM-based programming models, recent work focus on how to convert a RAM-program (e.g. C program) into a circuit efficiently. One big challenge is how to efficiently support secure computation over a random-access-machine (RAM) model. In [4], a customized construction featuring ORAM was presented to achieve better asymptotic performance for the binary search problem.

My work [6, 8] generalized this technique to support arbitrary RAM-model secure computation. Our idea is to provide a RAM-model programming language, which can be compiled to a highly efficient circuit. The key challenge is to achieve instruction trace and memory trace obliviousness during the compilation. To enforce obliviousness, I designed security type systems, which accept efficient program implementations (e.g. using ORAM for random-access patterns). Using our language, one PhD student spent one day to implement a secure a privacy-preserving matrix factor algorithm, which used to cost 5 researchers 4 months to implement in 2013 [1, 10]. Further, evaluation shows that our implementation achieves $10\times$ to $20\times$ better performance (see [8] for more details). The source code is available at `https://github.com/oblivm/ObliVMLang`.

Proposed work. Following this line of research, I am generalizing these existing work to allow general obliviousness analysis. I keep two design goals in mind during the development. First, the general framework should allow reasoning not only memory channels, but also other side channels within our interest. For example, my undergoing research can enforce protection of leakage through side channels including program counter channel, timing channel, and termination channel. The longer term goal will consider other side channels. Second, different from all prior work which considers ORAM as a black-box primitive, our framework should provide a core-calculus using which efficient ORAM protocols can be implemented, and their obliviousness can be enforced. Particularly, this goal requires reasoning about random numbers. My work solving these two challenges is under preparation for submission.

3.2 Reasoning Obliviousness for Parallel Programs

We extend our discussion with parallel programs, whose computation logic can be expressed in MapReduce-style distributed computation languages such as Spark, but the party providing data will pose security requirements about the computation. In this case, the cluster running the program may not be trusted. Although secure computation is a general solution to this class of problems, its computation cost is too expensive to be practical.

One recent promising work to enable privacy-preserving distributed computing is VC3 [11], which relies on Intel SGX secure processor to establish code integrity and data privacy on an untrusted party, while incurring little overhead. However, Intel SGX does not protect against memory address channels. Even replacing Intel SGX with an ORAM-capable security processor, this approach still leaks information through network IO. On the other hand, GraphSC [9] provides a general framework to hide all access patterns of computations which can be encoded into a graph algorithm using primitives proposed in [9]. This approach lacks the generality in the sense that some efficient algorithms may not be encoded in GraphSC's primitives.

Inspired by these work, optimizations are possible for certain computations. We consider the scenario where two parties each holding a list of numbers, and want to jointly compute the summation of all numbers. One naive approach is to leverage the secure computation solution from GraphSC to compute the summation. One improvement to this method is to ask each party to sum up their numbers before the secure computation. Further, we can adopt VC3's idea to replace SC with Intel SGX to perform the joint computation.

Proposed work. Inspired by this observation, I adopt a programming language design and system co-designed approach to build a secure distributed computing system. The system provides a language to allow programmer to specify local and joint computations. The language has an advanced type system that allows reasoning about the obliviousness with respect to network communication traces. We plan to supply an optimizer to allow programmers to specify the computation in an intuitive way without having to worry about optimization, and the optimizer can seek for an equivalent alternative specification with optimal performance using programs' running statistics collected by the system.

4. Evaluation Methodology

My main research results will be rigorously proved theorems to show that the security type systems can formally enforce programs' obliviousness in both dedicated application scenarios and the general framework. Secondary results are to show that my compilers can generate optimized code compared with the baseline approach which uses generic ORAM simulation-based approaches and GraphSC.

Experimental Setup. To evaluate my compiler for privacy preserving cloud computing, I implemented both the baseline approach which allocates all data in one ORAM bank, and an optimized approach that intelligently minimize the usage of ORAM banks. We evaluate the code using both an ORAM-capable secure processor prototype implemented in a FPGA. This evaluation shows that the optimized approach outperforms the baseline approach by a factor up to $10\times$, and never slows down. The detailed results can be found in [7].

For evaluation of my compiler for secure computation, I compared the circuits generated by my compiler, namely ObliVM, over prior state-of-the-art work [6]. We can observe an up to 10^6 speedup due to the new architecture of the ObliVM system, and among them a $2500\times$ speedup is attributed to the ObliVM compiler. More details about the evaluation setup and results can be found in our paper [8].

ObliVM project[1] has open sourced its compiler and Garbled Circuit backend, and has genomic and medical applications. ObliVM participated in the iDASH Secure Genome Analysis Competition[2], and won a champaign in one task.

For the parallel compiler, we plan to build the system along with the compiler and compare it with the existing oblivious programm framework GraphSC. We will take into account in our evaluation the following parameters: computation type (e.g. locally, joint); data distribution (balanced or skewed) and number of machines in the cluster. We will also evaluate the performance of our optimizer by comparing the performance of the output program of the optimizer and manually optimized programs.

References

[1] Personal communication with Nina Taft.

[2] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *JACM*, 1996.

[3] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proc. of STOC*, 1987.

[4] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In *Proc. of CCS*, 2012.

[5] C. Liu, M. Hicks, and E. Shi. Memory trace oblivious program execution. In *Proc. of CSF*, 2013.

[6] C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks. Automating efficient RAM-model secure computation. In *Proc. of S & P*, 2014.

[7] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *Proc. of ASPLOS*, 2015.

[8] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. In *Proc. of S & P*, 2015.

[9] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. Graphsc: Parallel secure computation made easy. In *Proc. of S & P*, 2015.

[10] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh. Privacy-preserving matrix factorization. In *Proc. of CCS*, 2013.

[11] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud using SGX. 2015.

[12] X. Zhuang, T. Zhang, and S. Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. 39(11):72–84, 2004.

[1] `http://www.oblivm.com`

[2] `http://www.humangenomeprivacy.org/2015/`