Languages for Oblivious Computation

Michael Hicks
University of Maryland

1 OBLIVIOUS COMPUTATION

Cloud computing allows users to delegate data storage and computing needs to cloud service providers. Doing so relieves users from the need to purchase and maintain their own computing infrastructure, but requires sharing potentially sensitive data with the provider. Researchers have been exploring how to mitigate the risk of doing so by developing *privacy preserving* computing technology. The idea is to employ hardware and/or software that can compute a function $y = f(x_1, x_2, ..., x_n)$ *obliviously*, meaning that despite producing an answer, the service provider learns nothing about $x_1, ..., x_n$ or y in the process. Such technology effectively implements a secure abstract machine that receives encrypted inputs, computes the requested function using encrypted memory, and returns an encrypted result, which the client can decrypt. Such an abstract machine might employ cryptographic algorithms and/or secure processors (e.g., FHE [1] or Intel SGX[1]).

While a secure abstract machine stops an attacker from reading sensitive values directly, it does not defeat an attacker who can *infer* such values from a computation's *side channels*, such as its patterns of memory accesses or instruction timings. Indeed, a cloud provider could easily measure such patterns.

A countermeasure is to augment the abstract machine to store code and data in oblivious RAM (ORAM) [2]. ORAM is a data structure that regularly changes the mapping between a data block's logical address and its physical address. While the abstract machine always knows the up-to-date mapping, the adversary does not, and as a result the address trace is indistinguishable from a random sequence. While secure against a snooping adversary, ORAM unfortunately incurs a substantial slowdown in practical situations: each read/write requires additional operations (to update the mapping) that are polylogarithmic in the size of the memory.

2 COMBINING PL AND ORAM

While naïvely storing all code and data in ORAM is too slow, techniques from the language-based security community can significantly improve performance. In particular, notice that ORAM is only necessary when the address trace depends on the values of secrets. Code snippets like `x[h]` are dangerous since the address observed when reading from array `x` reveals something about the secret `h`. But code like the following is safe:

```
for (int i=0; i<n; i++) { m = m + x[i]; }
```

Assuming that `n` is not a secret, the stream of addresses read is entirely determined, i.e., it does not depend on any secret. As such, while it might make sense to put `x` in *encrypted* RAM if its contents are secret, putting `x` in ORAM is unnecessary.

Obliviousness by typing. In a paper published at CSF'13 [4] we developed a type system for a language in which arrays can be stored in either encrypted RAM or oblivious RAM. Type correctness ensures *memory trace obliviousness* (MTO), meaning that the address trace does not depend on secret inputs. A program like the loop above would be type-correct even when `x` is not in ORAM, but a program like `x[h]` requires `x` to be in ORAM (if `h` is secret).

Ghostrider. We developed our basic approach into hardware/-software co-design called Ghostrider [3]. On the hardware side, we developed a coprocessor to implement a state-of-the-art ORAM algorithm, and a processor for an extension of the RISC-V instruction set. On the software side, we developed a compiler that uses the ideas from our basic approach to allocate data to encrypted RAM or ORAM. We developed a type system for the generated assembly code, thus ensuring MTO without needing to trust the compiler. On both the real hardware and a simulator for a more full-featured architecture we found we could achieve an order-of-magnitude speedup for many common programs, as compared to putting all data in ORAM.

3 ONGOING WORK

Ghostrider treats ORAM as a trusted primitive. In ongoing work we are developing more fundamental language mechanisms with which one can *implement* ORAM, as well as optimized *oblivious data structures* (ODSs) [5]. For example, while we could program an *oblivious queue* as a queue stored in ORAM, we can improve performance by applying ideas from general ORAM implementations to queue implementations, directly. A core idea in ODS implementations is the controlled use of randomness to obfuscate the logical-to-physical mapping. We are developing a language called L_obliv that ensures these mechanisms are used safely. In particular, type correctness ensures *probabilistic* MTO, meaning that the *distribution* of visible events does not depend on secrets. L_obliv is expressive: it can type check state-of-the-art ORAM and ODS implementations.

REFERENCES

[1] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme.* Ph.D. Dissertation.
[2] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *J. ACM* (1996).
[3] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* **Best Paper**.
[4] Chang Liu, Michael Hicks, and Elaine Shi. 2013. Memory Trace Oblivious Program Execution. In *Proc. of the Computer Security Foundations Symposium (CSF).* Winner of the 2014 NSA **Best Scientific Cybersecurity Paper** competition.
[5] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In *Proc. ACM Conference on Computer and Communications Security (CCS).*

[1] https://software.intel.com/en-us/sgx

PLAS'17, October 30, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-5099-0/17/10.
https://doi.org/10.1145/3139337.3139349