The goal of my research is to make privacy-preserving computation available to data sensitive applications. The big data era has dramatically transformed our lives; however, security incidents such as data breaches put sensitive data (e.g. photos, identities, genomes) at risk. To protect users’ data privacy, there is a growing trend to build privacy-preserving computing systems, which enables computation over two or more parties’ sensitive data, while revealing nothing more than the results to the participating parties. Conceptually, privacy-preserving computing systems leverage cryptographic techniques (e.g. secure multiparty computation) and trusted hardware (e.g. secure processors) to instantiate a “secure” abstract machine consisting of a CPU and encrypted memory, so that an adversary cannot learn information through either the computation within the CPU or the data in the memory. Unfortunately, evidence has shown that, side channels (e.g. memory accesses, timing, and termination) in such a “secure” abstract machine may potentially leak highly sensitive information including cryptographic keys that form the root of trust for the secure systems.

I conduct synergistic research to bridge cryptography and programming language techniques to address this problem. My PhD research broadly expanded the investigation of a research direction called trace oblivious computation, where I employ programming language techniques to prevent side channel information leakage. My research philosophy is “Theory Meets Practice”. On the theoretical side, I developed novel security type systems to verify if a program is free of side channel leakage, or a.k.a. trace oblivious. On the practical side, I built a secure hardware-based cloud computing platform, called GhostRider and a cryptography-based secure computation framework, called ObliVM. While previous work also studied mitigating side channels, especially memory access traces, using Oblivious RAM (ORAM), they stored all data into ORAM, and thus incurred a moderate overhead for each memory access. My approaches enabled accurate program analysis, and thus allowed more efficient oblivious implementations than previous work.

Beyond my main research area at the boundary of programming language, cryptography, and security, I also work on distributed systems and machine learning, to make privacy-preserving computing systems scalable and customized for data analytical applications. I am building a unified data analytical platform, called Helio, to manage different data sources and computing resources, with respect to the different security policies imposed by data and resource providers. Helio is expressive and easy-to-use, and enables sensitive data sharing in the context of healthcare and sensor network. In the future, my vision is to make privacy-preserving computing easy to achieve in all applications involving sensitive information.

1 Memory Trace Obliviousness: Theory and Practice

The first problem I studied is how to leverage trusted hardware to build a privacy-preserving computing system in the presence of physical attackers. Particularly, a physical attacker can observe all communications outside the processor, including memory addresses and data transmitted over the bus. An idealized solution to defend such an attacker is to employ a secure processor to guarantee all data transferred out of the processor were encrypted. Several candidates (e.g. Trusted Platform Module) were proposed to instantiate the secure processor. However, they could not protect memory access patterns.
My research thus focused on mitigating leakage through memory access traces. Cryptographic Oblivious RAM (ORAM) protocols provided a generic solution to build secure ORAM processors, while paying the cost of a poly-logarithmic overhead (in the memory size). My collaborators and I observed that, a program with memory access pattern not depending on secret inputs could avoid paying the ORAM overhead. From this observation, I developed a theory to describe programs that enjoy the memory trace oblivious property [5]. Our paper won the 2013 NSA Best Scientific Cybersecurity Paper Award.

Based on this theoretical result, my collaborators and I built a hardware-compiler co-designed system, called GhostRider [4]. GhostRider customizes a RISC-V ORAM processor, and includes a compiler to take advantage of the customized hardware. GhostRider serves as a trusted-third party using remote attestation for integrity, and ORAM for privacy.

GhostRider has many desirable security properties. First, GhostRider’s type checker for assembly code enforces that well-typed assembly code is memory trace oblivious, which also implies that the code is free of termination channel leakage. Second, GhostRider eliminates the timing channel leakage entirely by using deterministic timing instructions in the hardware. Third, GhostRider disables the hardware cache in its secure processor to eliminate the cache channel leakage.

GhostRider is easy-to-use in the sense that programmers need only annotate the input variables of a function while leaving the rest as a subset of standard C code. The compiler will try to infer the minimum security labels for all intermediate variables. GhostRider is also performant. Although GhostRider removes the implicit-cache, it implements a software-controllable scratchpad instead. The GhostRider compiler takes advantage of the scratchpad to simulate a memory trace oblivious software cache, so that GhostRider programs still enjoy the benefit of caching. Evaluating several benchmark programs on FPGAs showed that GhostRider achieved up to $10 \times$ speedup over previous oblivious systems such as Phantom [11]. The paper [4] won the Best Paper Award at ASPLOS 2015.

2 Practical RAM-model Secure Computation

For applications where no trusted hardware is available, I focused on the cryptography-based privacy-preserving computing solution, secure computation (SC) [7, 9]. SC has been studied for more than three decades in the cryptography community, but most previous results employ a circuit-based computation model, which is far from the commonly adopted Random Access Machine (RAM) based model. Prior work on compiling RAM-based programs into circuits incurs a significant overhead (i.e. linear in the memory size) when compiling a memory access with a secret address. As a result, such systems are not really suitable for many applications.

To fill the gap, my collaborators and I developed ObliVM [7, 9] as a programming framework to enable efficient translation from a RAM-based program into a circuit-based secure computation protocol using ORAM. ObliVM has three design goals: (1) expressiveness: ObliVM should be flexible enough to allow expert cryptographers to implement efficient secure computation protocols (e.g. ORAM); (2) easy programmability: application developers should be easy to build applications without too much knowledge about cryptography; and (3) security: the security of an ObliVM program requires not only memory trace obliviousness, but also no information leakage through the observation of the executed instructions.

I balanced these design goals in the ObliVM programming language by enabling programming abstractions. Briefly speaking, the ObliVM language allows expert programmers to develop optimized abstractions, such as MapReduce and data structures, which application developers can use despite little cryptography knowledge. ObliVM can greatly improve developers’ efficiency. For example, the secure ridge regression implementation [12] took the authors 15 person-weeks to develop, while its ObliVM counterpart took only 2 person-hours, and achieved $3 \times$ smaller circuits sizes. I have open-sourced the tools at http://www.oblivm.com. ObliVM [9] won the First Place in the Best Applied Cyber Security Research Paper Competition at CSAW 2015.
3 A Core Calculus for Oblivious Computation

While working on ObliVM, I became interested in the theoretical question whether we could automatically prove the obliviousness of an ORAM implementation itself using a type system. Since the best known deterministic ORAM incurs quadratic overhead, I needed to develop a security type system to handle randomness and probabilities. To deepen my understanding of oblivious algorithms, I worked on oblivious data structures [13], and parallel ORAM algorithms [2]. Through these works, I understood that a typical ORAM implementation relies on two important properties of random numbers to achieve security: (1) a secretly generated random number can be revealed to the adversary once without sacrificing security; and (2) to guarantee all secret random numbers are sampled from the same distribution, operations between dependent random numbers should be forbidden.

Based on these observations, I developed a core calculus, called $\lambda_{obliv}$ [6] to reason the two properties. First, $\lambda_{obliv}$ employed an affine type system to keep track of random numbers. Intuitively, random numbers can be considered as scarce resources, and the affine type system can syntactically guaranttee that each generated random number has at most one copy during execution. Second, I introduced the concept of probability regions so that all values within the same probability region are independent from each other. To enforce the second requirement, the type system allows only operations among values in the same region. Using this calculus, I was the first to implement efficient practical ORAM algorithms, such that a type system can guarantee the obliviousness of the implementation.

4 Toward Secure Collaborative Data Analytics

An important application for privacy-preserving computing systems is to enable collaborative data analytics among multiple parties. Therefore, I became interested in building a unified platform to allow performing data analytics using different data sources and computing resources, both of which are attached with security policies. To this end, I built Helio [8] to instantiate such a platform. Helio was built on top of a popular data processing system, Apache Spark\(^1\), to smooth users’ learning curve.

Besides the computation language, Helio provides a policy language, called HPL, to allow data providers to attach policies to data sources. HPL policies are similar to legal policies in English to ease non-programming experts (e.g. legal team) to understand and verify them. These policies are transparent to programs, and Helio enforces the policies automatically. I developed a core-calculus to ensure the soundness of HPL semantics and the enforcement mechanisms.

Helio is a distributed system managing computing resources from multiple parties. The security policies restrict data from being reallocated freely, and therefore incur scheduling challenges. I designed a simple scheduler, which yields $2.6 \times$ to $6.4 \times$ better performance than a baseline.

Helio can be deployed under different trust models. When the computing infrastructure is trusted, Helio can prevent malicious computation to steal participating parties’ sensitive data. When the computing infrastructure is not trusted, Helio can be extended to employ Intel SGX [1] to protect against software attackers, or GhostRider and ObliVM for physical attackers. We have demonstrated that Helio is practical for use cases such as healthcare and sensor networks.

5 Future Directions

I am fascinated by challenging security and programming language problems. In the following, I will discuss several future directions that I am interested in.

\(^1\)http://spark.apache.org/
5.1 Verified ORAM Processor

I have demonstrated that the calculus $\lambda_{\text{obliv}}$ is expressive enough to implement an efficient and well-typed ORAM algorithm. A more challenging but promising problem is to type check the ORAM implementation in secure processors, such as GhostRider [4]. There are two general challenges. On one hand, there is an expressiveness gap between $\lambda_{\text{obliv}}$ and popular hardware programming languages, such as Verilog [14]. On the other hand, the type system for $\lambda_{\text{obliv}}$ enforces instruction trace obliviousness, which may be stronger than necessary for hardware ORAM implementations. Despite these differences, keeping track of randomness during execution is still essential to reason the memory trace obliviousness for ORAM processors.

5.2 Secure System for Big Data

My current work on Helio opens an exciting area to manage heterogeneous data, computation resources, and security policies in a unified platform. I am particularly interested in exploring two directions in this field. First, it is appealing to make security policy enforcement transparent to data analysts. Achieving this goal is challenging. A naive solution is to remove all information forbidden by the policies, but doing so may also reduce the utility of data analytical programs. The current solution in Helio preserves more utility for many real policies, but may degrade to the naive case when policies become more complex. I am interested in exploring tradeoffs between security and utility. Second, performance is important for big data processing. My previous work [3, 10] has exploited programming analysis techniques to optimize MapReduce-style jobs. When considering the optimization problem in the security context, privacy policies pose new challenges. For example, standard optimization techniques such as fusion may not be applicable when security policies restrict on data locations. I will examine this problem in the context of real use cases.

5.3 Contract Language for Cryptocurrency

A trending security and cryptography application area is cryptocurrency and smart contracts. Cryptocurrency systems such as Bitcoin and Ethereum provide a decentralized transactional database called the blockchain to enable cryptography-enforced smart contracts. The smart contract technologies have enabled many promising practical applications.

The development of programming languages to write smart contracts mostly focuses on expressiveness, with less emphasis on security. Evidence has shown that it is easy for inexperienced programmers to implement contracts in a way such that one party of the contract can gain advantages over the others. In particular, by observing the information that one party commits to the blockchain, other parties may strategically choose to abort the execution of a contract, so as to avoid the bigger loss that will be incurred otherwise. I consider this as a fairness problem, and I am interested in studying how to guarantee (automatically) that a contract properly incentivizes all participating parties to faithfully execute the contract.

References


