Full Presentation: Migration to the Cloud made Safe and Secure

Ken Eguro Kaushik Rajan Ravi Ramamurthy Kapil Vaswani Ramarathnam Venkatesan
Microsoft Research
eguro,krajan,ravirama,kapil,venkie@microsoft.com

1. The Problem

In the last few years, cloud computing has evolved from a buzzword to a critical infrastructure component of many enterprise and consumer services. The cloud provides virtually limitless compute, storage and network resources at low cost, allowing services to scale on demand. The cloud absolves organizations from managing IT infrastructure, and allows them to focus on their core competencies.

However, the benefits of cloud computing do not come for free; building and running applications for the cloud comes with significant challenges. Arguably the most significant challenge is security. By their very nature, applications deployed on a public cloud expose a larger attack surface when compared to their in-house counterparts. Applications on the cloud are hosted in a multi-tenant environment, where they share physical resources such as memory, disk, network and CPU. This model, which is key to cloud providers achieving benefits of scale, enables a variety of attacks from co-located malicious applications. Another security threat are is the cloud operator, who can both observe and tamper with an application's execution. These limitations have precluded the migration of security sensitive applications to public cloud platforms, forcing organizations to consider more expensive and less scalable alternatives such as the private cloud.

Data encryption. One approach for guaranteeing security of sensitive data on public cloud platforms is encryption (e.g. using private key encryption). Encryption ensures that data at rest is never seen in plain text on untrusted machines (such as the application and database server). Therefore, data is protected from direct attacks such as theft and tampering, though the possibility of side channel attacks and frequency attacks remain. Most contemporary storage systems offer encryption and key management as core services. For example, SQL Server supports various encryption schemes, both granularity of individual columns, or the entire database.

Encryption and application compatibility. Ideally, one would like encryption to be a *semantics preserving* i.e. encryption should preserve the behavior of applications written to work with unencrypted data . However, efficient, semantics preserving encryption schemes (i.e. homomorphic encryption [2]) remain elusive. Most encryption schemes used in practice are efficient but significantly limit the kind of computations that can be performed directly on encrypted data while preserving semantics. For example, one way hash functions and deterministic encryption schemes permit equality checks to be performed directly on encrypted data while preserving semantics. Therefore, database operations such as equi-joins and grouping that rely on equality checks can be performed without requiring the data to be decrypted. Strong encryption schemes such as non-deterministic encryption do not permit

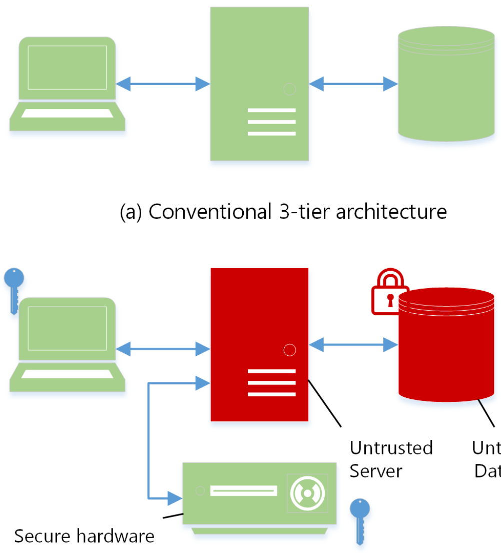
(a) Conventional 3-tier architecture
Untrusted Server
Untrusted Database
Secure hardware
(b) Architecture post encryption

Figure 1. Architectures of a typical 3-tier application pre and post encryption

even equality checks. In general, data encryption is not semantics preserving, and has a huge bearing the rest of the application stack.

For example, consider a typical 3-tier application with a browser based or native client, a middle tier and a data layer consisting of databases, stored procedures, views and other data access components (Figure 1 (a)). Assume that the application is written without considering encryption. Encrypting parts of the database can break some or all of these layers, and significant rewriting is required to restore the application's functionality. First, the application must be rewritten to manage encryption keys. For sensitive data, it is desirable to store encryption keys with the client and not in the middle tier or database (which would defeat the purpose of encryption). Therefore, client side logic must be rewritten to encrypt data before it is passed to the middle tier, and vice versa.

More importantly, the business logic in the middle tier and the data layer must be carefully analyzed to detect computations that are *incompatible* with encryption. For example, it is not possible to sort or compute aggregations over encrypted columns (even when using one way hash functions) while preserving semantics. Therefore, any such computation must be pushed to the client. If the middle tier is responsible for rendering the UI and the rendering logic depends on encrypted data values, the rendering logic must be pushed to the client. The client logic must be rewritten to first decrypt the data (since it has access to keys), perform the necessary operations, and optionally encrypt the results and send them back to the middle tier. Therefore, business logic that was originally centralized and easy to write and reason about, must now be distributed across multiple layers.

The problem of distributing the application's logic is compounded if a trusted components such as a secure co-processors or FPGA based secure programmable hardware [1] is added to the mix (Figure 1 (b)). These components have access to keys and can perform a limited class of computations on encrypted data securely (i.e. without revealing plaintext) even though they are physically hosted in an untrusted envrionment. Pushing computations to such devices lowers communication costs since sensitive data no longer needs to be shipped all the way to the client if the computation is supported by secure hardware.

Today, the process migrating applications post-encryption is performed manually. The process requires a thorough understanding of the entire application's logic and dataflows between applications, if any. It also requires developers with expertise in multiple languages and frameworks. The rewriting process is error prone, and therefore, multiple rounds of testing are required to ensure security and correctness.

Another challenge faced during the process of migrating an application is the trade-off between encryption and performance. Strong encryption schemes (such as non-deterministic encryption) provide stronger security guarantees at the cost of computations that can be performed without decryption. Therefore, both security experts and developers must compare different encryption policies in terms of their impact on the performance and the security guarantees they provide. This is challenging because applications routinely have large databases with many tables and columns and the search space of encryption policies can be extremely large.

Tool support for application migration. As described above, the process of migrating data and computation to untrusted hosts is akin to low level assembly programming. We believe that for the full potential of the cloud to be realized, developers must be supported with tools that automate the process of migrating applications while guaranteeing correctness, performance and the desired level of security. Specifically, we envisage a development environment where security experts or application developers state their security requirements declaratively, and the application is automatically migrated to meet the given security requirements while preserving behavior with minimal manual effort.

2. Role of programming language research

Programming language techniques have a key role to play in isolating developers from the challenges of guaranteeing security. For example, static analysis techniques such as type systems, data flow analysis and compiler optimizations can play an important role in automating the following tasks for a large class of applications.

- **Checking application compatibility.** The first problem that arises once an encryption policy has been enforced is to check if the rest of the application stack is affected. More formally, given a program p, we would like to check whether its behavior when all state is plain text is equivalent to its behavior when some parts of state are encrypted. For example, consider the following SQL query.

  ```
  SELECT c.Name, c.Address
  FROM dbo.Customer c
  JOIN dbo.Sales s ON s.CustomerId = c.CustomerId
  ```

  The behavior of this query when all columns are plain text is equivalent to its behavior when the columns `CustomerId` in the tables `Sales` and `Customer` are encrypted using a deterministic encryption scheme, and the columns `Name` and `Address` are in plain text.

- **Secure rewriting.** The secure rewriting problem arises for applications that do not satisfy the above equivalence criteria. Such programs must be rewritten to identify computations that are not compatible with the given encryption policy. Such computations must be rewritten to decrypt and encrypt data. Specifically, given a program p, we wish derive a program p' such that the behavior of p when the initial state is plain text is equivalent to the behavior of p' when some parts of the initial state are encrypted. The program p' is assumed to have access to keys and may use routines to encrypt and decrypt parts of state as required.

  Consider the query above. If columns `Name` and `Address` are encrypted, the query must be rewritten as follows. The routine `DECRYPT` decrypts the column using the given key.

  ```
  SELECT DECRYPT(k, c.Name), DECRYPT(k, c.Address)
  FROM dbo.Customer c
  JOIN dbo.Sales s ON s.CustomerId = c.CustomerId
  ```

- **Program partitioning.** The above rewriting is incomplete because it assumes that the host has access to keys. In a real deployment, keys are accessible to clients and trusted hardware but not to the database and application servers. Therefore, we require that the application be partitioned between different hosts. Formally, given a program p, we wish to derive a program p' such that the behavior of p when all state is plain text is equivalent to the behavior of p' when some parts of the initial state are encrypted. Further, we require that p' consist of two components, a *trusted* component that has access to key and may use encryption and decryption routines, and an *untrusted component* that does not have access to keys but has access to state. We also require that the component delegated to trusted hardware should only run computations that are supported by the hardware.

Our current work focuses on designing static analysis techniques that address some of these challenges for a small part of the application stack, namely stored procedures written in T-SQL. We first formalize the correctness and security guarantees of stored procedures deployed on an untrusted host. We propose a simple type system for T-SQL with one base type for each type of encryption, record types and function types. We define a sub-typing relation between types; the sub-typing relation models conversion of values from one encryption type to the other. For example, we consider plain text to the a sub-type of deterministic encryption since plain text values can be converted to deterministically encrypted values using appropriate encryption routines. The problem of checking application compatibility can be reduced to type inference over this type system. Furthermore, secure rewritings for incompatible applications can be generated by automatically inserting *coercions*. We then prove that the rewritings we generate are both secure and behaviorally equivalent to the original stored procedures. However, we recognize that our work represents just initial steps in this space , and a much larger research effort is required to realize the goal of transparent migration of applications to the cloud.

References

[1] R. V. Ken Eguro. FPGAs for Trusted Cloud Computing. In *International Conference on Field Programmable Logic and Application*, 2012.

[2] D. Micciancio. A first glimpse of cryptography's Holy Grail. *Communications of ACM*, 53(3):96, 2010.