Language support for efficient computation over encrypted data

Meelap Shah, Emily Stark, Raluca Ada Popa, and Nickolai Zeldovich
MIT CSAIL

1 PROBLEM

Many applications today store and operate on sensitive data. For example, Facebook keeps users' profile data and posts, Gmail stores users' emails, Amazon Silk observes user browsing patterns, and analytics pipelines operate on logs of user behavior. Moreover, applications are increasingly using third parties such as Amazon EC2 and S3 to provide computing and storage infrastructure, with the result that users' data is stored and operated on by third party servers. This forces end users to trust these third parties to not misuse their data, in addition to trusting the original application. Even if all these parties are benign, this increases the trusted computing base (TCB), and attackers can compromise *just one* of these services to gain illicit access to users' data. We would like to minimize the TCB and provide data confidentiality guarantees while allowing sensitive data to be sent into untrusted environments.

2 APPROACH

We propose solving the general problem of running computations over sensitive data in an untrusted environment using a combination of cryptography and automatic program partitioning. Advances in cryptographic schemes, such as fully homomorphic encryption [3], allow arbitrary computations on encrypted data while leaking no information to the server. Unfortunately, computations using fully homomorphic encryption are orders of magnitude slower than computations over plaintext data.

Another approach is to focus on specialized encryption schemes that efficiently enable specific computations over encrypted data, and in some cases leaking a limited amount of information to the server. For example, CryptDB [5] showed how most computations supported by SQL database queries can be performed over encrypted data: deterministic encryption schemes allow equality checks, homomorphic schemes allow additions [4], and order preserving schemes allow comparisons [1]. Each of these schemes offers different levels of confidentiality—the ability to perform some operations on ciphertext usually comes at the cost of leaking some number of bits about the ciphertext, such as a few high bits for an order-preserving scheme—but never reveals enough information to recover the plaintext. To reveal as little information to the server as possible, it is important to use the most secure scheme that supports all of the necessary operations.

We wish to extend this idea so that application logic can also be made to execute over ciphertext. Since not all operations that applications perform will be supported by some encryption scheme, we will partition the program into two components. The first will consist purely of operations that we know how to run over ciphertexts, and can run on an untrusted server, and the second will consist of all remaining operations, and must run on a trusted system (such as the user's machine). The trusted partition will encrypt any data it sends to the untrusted partition, decrypt any results it receives back, and will not divulge the decryption keys. In practice, we expect that the application developer or administrator would indicate which pieces of data managed by an application are sensitive, and only encrypt those.

3 CHALLENGES

Program partitioning has been well studied, and we plan to use similar techniques [2, 6], but we must address several questions to prove our approach is feasible.

- How should we encrypt arguments to functions?
- When should we re-encrypt values (to perform different operations), and how to minimize re-encryption?
- How should we interact with remote objects?
- What applications are a good fit for these techniques?

We believe that with programming language support and program analysis, we may be able to build a tool to answer such questions for partitioning programs.

3.1 Intra-function Analysis

Given a program, our tool first determines what operations each function performs on each sensitive value. Next, our tool checks whether this set of operations is supported by any of the encryption schemes that it knows about. If so, that function can execute over encryptions of that sensitive value (in some cases, this may require changing the function, such as replacing the + operator with homomorphic addition). If our tool determines that all sensitive arguments to a function can be encrypted, the tool generates a wrapper for that function in the trusted partition, which first encrypts all arguments with the appropriate encryption schemes, and then executes that function on the untrusted partition over ciphertexts.

We believe that functional programming languages, such as Haskell, are a good fit for this kind of analysis,

since we can use the type system to express the computations supported by each encryption system, and use type inference to decide what computations can be performed over ciphertexts. In particular, we can define a type class for each class of computation that we know how to perform over encrypted data (corresponding to an encryption scheme). For example, Haskell's `Eq` type class would correspond to deterministic encryption, which allows equality comparisons, and Haskell's `Ord` type class would correspond to order-preserving encryption, which allows order comparisons. We would similarly define additional type classes to capture the sets of operations that we can support on other types of ciphertexts.

When some operation is performed on a value, the compiler infers that the value's type must be an instance of the corresponding type class. By enumerating all type classes to which a value's type must belong, the compiler will tell us exactly the set of operations performed on that value. Our tool does this type class inference for the sensitive inputs to each function, and if it recognizes an encryption scheme that supports all operations, it can encrypt that value. Successfully recognizing an encryption scheme for every sensitive input allows us to execute that function in the untrusted partition over ciphertexts.

Although Haskell's type classes provide a nice way to represent classes of computation, not all operations over ciphertexts that we would like to support can be represented this way. For example, there exist encryption schemes that support keyword search, but it is difficult to statically determine whether a piece of code is doing a search. One option is to ask the developer to use a specialized search function; another option is to explore analysis techniques for matching computation patterns to operations allowed by specialized cryptographic schemes.

3.2 Inter-function Analysis

As control flow passes between different functions, we may need to re-encrypt values under different schemes to support different sets of operations. For example, consider a simple application with two functions: `insert` adds a value to a list and returns the new list; `find` checks whether a given key is in the list. Since we prefer to use the strongest encryption scheme that supports the required operations, we would use a randomized scheme for `insert` and a deterministic scheme for `find`.

If both `insert` and `find` execute in the untrusted partition, we would need to pay a round trip back to the trusted partition to re-encrypt the output of `insert` so that it can be used as an input to `find`. However, if we construct a data flow graph, we may realize that we can use deterministic encryption for `insert` to save round trips. We would like to perform this type of data flow analysis statically so that we know how to encrypt values before remotely invoking a function.

3.3 Local Representation of Remote Objects

We would like to minimize the amount of data sent between the partitions. Consider the simple list application from above. In a naïve implementation, the entire list is transferred every time the trusted partition calls the untrusted partition to compute the result of either function. We would instead like the list to remain in the untrusted partition and return a *proxy object* to the trusted part. The actual object should be transferred to the trusted partition only when required. Moreover, the proxy object should appear to have the same type as the actual object so that it can be used transparently. This could be done with the help of programmer annotations, or using analysis techniques [2, 6].

4 Current Prototype

Our current prototype in Haskell begins to solve some of the challenges from §3. Currently we require the programmer to annotate all functions that should run in the untrusted partition. Given these annotations and a target module to split, our system first infers which inputs to the annotated functions are encryptable, and then generates RPC and encryption wrappers for them.

Rather than requiring the programmer to specify a partitioning and having our tool make a best effort at encrypting all data seen by these functions, we would instead like the programmer to simply specify which pieces of data are sensitive. Our tool should then determine a partitioning so that these data are always encryptable before being sent to the untrusted partition.

Once we have a more complete prototype, we hope to analyze existing applications to find classes of programs that are a good fit for the proposed techniques. As one example, we hope that simple web applications perform little complex processing and mostly move data around.

References

[1] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *EUROCRYPT*, April 2009.

[2] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *SOSP*, October 2007.

[3] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, May–June 2009.

[4] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, May 1999.

[5] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*, October 2011.

[6] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *SOSP*, October 2001.