Programming on Encrypted Data

Dan Boneh, Kathleen Fisher,
John Mitchell
Challenge

- Cryptography gives us great tools
  - Secure function evaluation
  - Secret sharing
  - Homomorphic encryption

- How do we make these tools useful to a wider community of software developers?
Cloud computing

?
Specific challenge

Can we send computation to the cloud, without revealing program or data?
Specific challenge

Can we send computation to the cloud, without revealing program or data?

Helpful ideas

- Trusted local computation + untrusted cloud
  - Trusted preprocessing
  - Trusted post-processing
- Suffices to solve the problem for data
  - Program can be “Universal Turing Machine”
    = Interpreter that receives ⟨program, input⟩
    (but consider other programs as examples too)
Homomorphic encryption

Compute function $f$ on encrypted data

$x$ $\rightarrow$ $\text{enc}(x)$ $\rightarrow$ $\text{enc}(f(x))$ $\leftarrow$ $f(x)$
Secret sharing

For $n,k$-secret sharing, secure if $<k$ servers collude.

\begin{align*}
\text{Compute share of } f(x) \text{ from share of } x \\
\text{Compute share of } f(x) \text{ from share of } x \\
\text{Compute share of } f(x) \text{ from share of } x
\end{align*}
Basic software approach

- Develop Domain Specific Language (DSL)
  - Based on Haskell, pure functional language
- Support code that allows \textit{later} crypto choice
  - Homomorphic encryption
  - Secret sharing
- Key technical concept that gets us going
  - Haskell monads
  - Homomorphic encryption, secret sharing are both instances of monads, with similar operations
**Vision**

- **Domain-specific language embedded in Haskell**
- **Secret sharing monad (library) on “cloud” platforms**
- **Replicated Haskell runtime**
- **Debugging Environment (no crypto)**
- **Standard Haskell runtime**
- **Homomorphic encryption monad (library)**
- **Optimized custom crypto runtime**

- **Write Haskell code once using generic monad**
- **Execute code later on chosen platform**
Haskell

- Haskell is a programming language that is
  - Functional: general-purpose, strongly typed, higher-order, type inference, interactive and compiled use
  - Lazy: purely functional core, extensible w/ effects

- Designed by committee in 80’s and 90’s to unify research efforts in lazy languages.
  - Haskell 1.0 in 1990, Haskell ‘98, Haskell ongoing
Higher-Order Functions

Functions that take other functions as arguments or return a function as a result

Common Examples:
- Map: applies argument function to each element in a collection
- Reduce: takes a collection, an initial value, and a function, and combines the elements in the collection according to function.

```
list = [1,2,3]
r = foldl (\accumulator i -> i + accumulator) 0 list
```

Google uses Map/Reduce to parallelize and distribute massive data processing tasks [Dean, Ghemawat, OSDI 2004]
(Haskell had these functional programming concepts long before Google)
Monads

- General concept from category theory
  - Adopted in Haskell for I/O, side effects, …

- A monad consists of:
  - A type constructor \( M \)
  - A function \( \text{bind} :: M a \rightarrow (a \rightarrow M b) \rightarrow M b \)
  - A function \( \text{return} :: a \rightarrow M a \)

- Plus:
  - Laws about how these operations interact
Monad Examples

- **Error handling**
  \[ M(A) = A \cup \{\text{error}\} \]
  - Add a special “error value” to a type
  - Define sequencing operator “;” to propagate error

- **Information-flow tracking**
  \[ M(A) = A \times \text{Labels} \]
  - Add information flow label to each value
  - Define “;” to check and propagate labels

- **Homomorphic encryption**
  \[ M(A) = \text{HomEnc}(A) \]
  - Represent values by encrypted values
  - Define “;” to homomorphically apply next function

- **Secret sharing**
  \[ M(A) = \text{Shares}(A) \]
  - Represent value by list of shares
  - Define “;” to apply next function to shares

*Can write code to compute on A, but run it using M(A)*
Monad “do” and “bind” notation

- The special notation

\[
\text{do \{v1 <- e1; e2\}}
\]

is “syntactic” sugar for the ordinary expression

\[
e1 >>= \lambda v1 \rightarrow e2
\]

where >>= (called bind) sequences actions

\[
(>>=) :: M a \rightarrow (a \rightarrow M b) \rightarrow M b
\]

- The value returned by the first action needs to be passed to the second; hence the 2\textsuperscript{nd} arg to >>= is a function (often an explicit lambda).
Monad feature of Haskell

Define monad for each type of encrypted data
- Secret sharing, executed on independent platforms
- Homomorphic encryption
- “Identity” monad with no encryption (for testing)

Conventional imperative notation
- Haskell code for computing over monadic values looks like standard imperative code

Interpreted using operations of monad
- The sequence operator “;” of the monad composes functions on encrypted data
Homomorphic encryption monad

Homomorphic encryption provides
- For all encryptable types S and T, a map
  \( (S \rightarrow T) \rightarrow \text{Encrypt}(S,k) \rightarrow \text{Encrypt}(T,k) \)
  that allows a function on public data to be applied to encrypted data, producing encrypted results

Haskell monadic programming requires
- For all types S and T, a map
  \( \text{Encrypt}(S,k) \rightarrow (S \rightarrow \text{Encrypt}(T,k)) \rightarrow \text{Encrypt}(T,k) \)
  that is used as the ";" for programming

Mismatch
- Resolved using circular-secure encryption (next slide)
Circular-secure encryption

- **Proxy re-encryption**
  
  \[
  \text{Encrypt}(S,k_1) \rightarrow \text{Encrypt}(\text{Encrypt}(S,k_1),k_2) \\
  \quad \rightarrow \text{Encrypt}(S,k_2)
  \]

- **Circular-secure proxy re-encryption**
  
  \[
  \text{Encrypt}(\text{Encrypt}(S,k),k) \rightarrow \text{Encrypt}(S,k)
  \]

- **Homomorphic encryption provides**
  
  \[
  \text{Encrypt}(S,k) \rightarrow (S \rightarrow \text{Encrypt}(T,k)) \\
  \quad \rightarrow \text{Encrypt}(\text{Encrypt}(T,k),k)
  \]

- **Monad condition satisfied**
  - Compose result from homomorphism property with map associated with proxy re-encryption
Additional technical issue

Homomorphic encryption provides
- For all *encryptable* types S and T, ...

Haskell monadic programming requires
- For all types S and T, ...

Problem
- Haskell assumes every type can be monadic
- Encryption applies only to numbers, pairs, ...
- Homomorphism defined for functions representable by circuits
- How do we resolve this without complicating the design and use of the programming language?
Some building blocks

Information-flow analysis
- Functions on encrypted data must not leak confidential values through control flow, i.e., language must prevent implicit information flow
- There has been considerable research on static and dynamic information flow analysis

Language-based computational complexity
- Functions on encrypted data must terminate and in some cases must have poly-size circuits
- S. Cook and students have produced language-based characterizations of complexity classes that yield type systems characterizing polynomial time
Current activities

- Define core expression language
  - Two types of integers: secret, public
  - Operations: add, multiply, if-then-else, ...

- Provide two semantics
  - Trusted sequential execution
  - Distributed execution on shares of secrets

- State and prove basic results
  - Compare sequential and distributed execution
Basic theorems (in progress)

Expressiveness

- For any computable function \( f: \text{Int} \rightarrow \text{Int} \), with computable time bound \( t \), there is a program \( P: \text{PInt} \times \text{SInt} \rightarrow \text{SInt} \) with \( P(t(|x|), x) = f(x) \)

Secrecy

- At each step in any distributed computation on shares of a secret input, each node has learned only shares of the secret intermediate results computed by the corresponding centralized trusted computation
Summary

- Exciting crypto possibilities
  - Homomorphic encryption
  - Secret sharing

- Current work on languages, tools for programming on encrypted data

- Leverage
  - Functional programming, monad concept
  - Program semantics, equivalence proofs
  - Related work on secure multiparty computation, crypto programming languages, information flow, …