In 2014, I taught smart contract programming to undergraduate students.
Smart contract programming: you are programming a distributed system
Smart contract
Smart contract
Rock-paper-scissors: “Hello World” for smart contract programming
Rock-paper-scissors: “Hello World” for smart contract programming

Smart contract
Rock-paper-scissors: “Hello World” for smart contract programming
Rock-paper-scissors: “Hello World” for smart contract programming
Rock-paper-scissors: “Hello World” for smart contract programming
def input(choice):
    if self.storage["player1"] == msg.sender:
        self.storage["p1value"] = choice
        return(1)
    elif self.storage["player2"] == msg.sender:
        self.storage["p2value"] = choice
        return(2)
    else:
        return(0)
Is this secure?

def input(choice):
    if self.storage["player1"] == msg.sender:
        self.storage["p1value"] = choice
        return(1)
    elif self.storage["player2"] == msg.sender:
        self.storage["p2value"] = choice
        return(2)
    else:
        return(0)
def input(choice):
    if self.storage["player1"] == msg.sender:
        self.storage["p1value"] = choice
        return(1)
    elif self.storage["player2"] == msg.sender:
        self.storage["p2value"] = choice
        return(2)
    else:
        return(0)
Use a **commit-and-open** protocol

... commitment should be non-malleable
Commit phase

Smart contract
Open phase

Smart contract
Lesson learned:

Distributed programming often involves *cryptography*.

Even the “Hello World” for distributed programming is hard!
Can we let ordinary programmers program cryptography without programming cryptography?
Our dream:

- Programmer gives a high-level specification with **security annotations**
- **Synthesize** an **efficient** cryptographic protocol
Two Challenges

- Cryptography speaks the *circuits*, not programs
  - e.g., multi-party computation, zero-knowledge proofs

- Choosing the right and most efficient *cryptographic primitive*
Two Challenges

Cryptography speaks the **circuits, not programs**
* e.g., multi-party computation, zero-knowledge proofs

Choosing the right and most efficient cryptographic primitive
Compiling programs to multi-party computation (MPC) protocols

Joint work with Chang Liu, Michael Hicks, and others
Example: Joint Clinical Study
MPC: learn **only the outcome** and nothing else
MPC: learn only the outcome and nothing else
Security: as secure as using an ideal functionality
program for the ideal functionality

Our compiler

Efficient MPC implementation
Programs

Dynamic memory accesses

Circuits

Static wiring
**Binary search:** access patterns depend on query

```python
func search(val, s, t)
    mid = (s + t)/2
    if val < mem[mid]
        search (val, 0, mid)
    else search (val, mid+ 1, t)
```
Naive idea 1 (secure but inefficient)

Use a **linear-scan circuit** to implement every memory access
Naive idea 2 (efficient but **insecure**)

Each step of the computation is a circuit, each circuit reads and writes memory
Oblivious RAM

Memory accesses do NOT leak information
Oblivious RAM

- Memory accesses do NOT leak information
- Each step $\Rightarrow$ poly log circuits
Signal, a private messaging app with >40 million monthly active users, runs the Path ORAM algorithm!
Naive idea: Put everything in ORAM
In practice, not all data must be placed in ORAM.

Accesses that do not depend on secret inputs need not be hidden.
Example: FindMax

```java
int max(public int n, secret int h[]) {
    public int i = 0;
    secret int m = 0;
    while (i < n) {
        if (h[i] > m) then m = h[i];
        i++;
    }
    return m;
}
```
int max(public int n, secret int h[]) {
    public int i = 0;
    secret int m = 0;
    while (i < n) {
        if (h[i] > m) then m = h[i];
        i++;
    }
    return m;
}
for(int i=1; i<n; ++i) {
    int bestj = -1;
    for(int j=0; j<n; ++j)
        if(!vis[j] && (bestdis < 0 || dis[j] < bestdis))
            bestdis = dis[j];

    vis[bestj] = 1;
    for(int j=0; j<n; ++j)
        if(!vis[j] && (bestdis + e[bestj][j] < dis[j]))
            dis[j] = bestdis + e[bestj][j];
}
for(int i=1; i<n; ++i) {
    int bestj = -1;
    for(int j=0; j<n; ++j)
        if(!vis[j] && (bestdis < 0 || dis[j] < bestdis))
            bestdis = dis[j];
    vis[bestj] = 1;
    for(int j=0; j<n; ++j)
        if(!vis[j] && (bestdis + e[bestj][j] < dis[j]))
            dis[j] = bestdis + e[bestj][j];
}
T Stack@m<T>.Op(T operand, int1 op) {
    T ret;
    if (op == 1) { // POP
        StackNode@m<T> r = this.poram
            .readNRemove(this.size, this.root);
        this.root = r.next;
        this.size = this.size - 1;
        ret = r.data;
    } else { // PUSH
        StackNode@m<T> node =
            StackNode@m (next = this.root, 
                data = operand);
        this.root = RND(m);
        this.size = this.size + 1;
        this.poram.write(this.size, 
            this.root, node);
    }
    return ret;
}
Automated, w/o ORAM

Hand optimized

ObliVM

Automated, w/ ORAM, no compile-time opt.

Hand optimized

1% difference between ObliVM and hand-optimized

Graph Algorithms

Machine Learning

Data Structures

Streaming Algorithms

ε = 0.001, r = 10

N = 2^20

V = 2^{16}

D = 10

#AND gates

Count Min Sketch

AMS Sketch

K-Means

kNN

Dense DFS

Dijkstra's algorithm

Stack

Map
Memory-trace oblivious type system
Memory-trace oblivious type system

Data sent to “low outputs” does not depend on secret inputs.

A program’s memory traces do not depend on secret inputs.
ObliVM: a programming framework for oblivious computation

Non-specialist programmer

Source program

Front-end Compiler

Oblivious representation

Back-end Compiler #1

Back-end Compiler #2

ORAM-based secure processor

Secure computation protocol
More details in our papers

Memory Trace Oblivious Program Execution. Joint with Chang Liu and Mike Hicks.


GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. Joint with Chang Liu, Michael Hicks, Austin Harris, Mohit Tiwari, Martin Maas.
Our related works

xjSNARK: Optimizing compiler for ZKP
Cool subsequent work by others

A Language for Probabilistically Oblivious Computation, POPL’20

By David Darais, Ian Sweet, Chang Liu, and Michael Hicks
Two Challenges

- Cryptography speaks the circuits, not programs
- Choosing the right and most efficient cryptographic primitive
Viaduct: automatically synthesizing cryptographic protocols

Joint work with Coşku Acay, Rolph Recto, Joshua Gancher, and Andrew C. Myers
What if the programmer doesn’t know which cryptographic primitive to use?
Implementing Shell with FLAM annotations

```scala
1   host alice: {A}
2   host bob : {B}
3
4   val n: {B ∧ A⁻} =
5   endorse (input int bob) from {B}
6   var tries: {A ⊓ B} = 5
7   var win: {A ⊓ B} = false
8   while (0 < tries ∧ !win) {
9       val guess =
10          declassify (input int alice) to {A ⊓ B⁻}
11   val tguess: {A ⊓ B} =
12       endorse guess from {A ⊓ B⁻}
13   win = declassify (n == tguess) to {A ⊓ B}
14   tries -= 1
15 }
16 output win to alice, bob
```
"Endorse" raises the integrity label

host Alice  // dealer
host Bob    // player

let shell = endorse (input Alice) to Bob
let valid = declassify (0 ≤ shell ≤ 2) to Bob
if valid:
  let guess = endorse (input Bob) to Alice
  let win = declassify (guess == shell) to Alice \ Bob
output win to Alice, Bob

Prevent dealer from changing shell
"Endorse" raises the integrity label

A ∧ B ←: private to A, trusted by A and B

host Alice // dealer
host Bob // player

let shell = endorse (input Alice) to Bob
let valid = declassify (0 ≤ shell ≤ 2) to Bob
if valid:
    let guess = endorse (input Bob) to Alice
    let win = declassify (guess == shell) to Alice ∨ Bob
output win to Alice, Bob
“Declassify” downgrades the privacy label

host Alice // dealer
host Bob // player

let shell = endorse (input Alice) to Bob
let valid = declassify (0 ≤ shell ≤ 2) to Bob
if valid:
    let guess = endorse (input Bob) to Alice
    let win = declassify (guess == shell) to Alice ∨ Bob
output win to Alice, Bob
“Declassify” downgrades the privacy label

\[(A \rightarrow \land B \rightarrow) \land (A \leftarrow \land B \leftarrow),\]

A and B can see, trusted by A and B

host Alice // dealer
host Bob // player

let shell = endorse (input Alice) to Bob
let valid = declassify \((0 \leq \text{shell} \leq 2)\) to Bob
if valid:
  let guess = endorse (input Bob) to Alice
  let win = declassify \((\text{guess} == \text{shell})\) to Alice \lor Bob
output win to Alice, Bob
“Declassify” downgrades the privacy label

host Alice // dealer
host Bob // player

let shell = endorse (input Alice) to Bob
let valid = declassify (0 ≤ shell ≤ 2) to Bob
if valid:
  let guess = endorse (input Bob) to Alice
  let win = declassify (guess == shell) to Alice V Bob
output win to Alice, Bob

Reveal the result
let shell = endorse (input Alice) to Bob
let valid = declassify (0 ≤ shell ≤ 2) to Bob
if valid:
    let guess = endorse (input Bob) to Alice
    let win = declassify (guess == shell) to Alice ∨ Bob
    output win to Alice, Bob
Synthesis: partitioning the program

host Alice // dealer
host Bob // player

let shell = endorse (input Alice) to Bob
let valid = declassify (0 ≤ shell ≤ 2) to Bob
if valid:
    let guess = endorse (input Bob) to Alice
    let win = declassify (guess == shell) to Alice ∨ Bob
output win to Alice, Bob
Synthesis: partitioning the program

```javascript
host Alice // dealer
host Bob   // player

let shell = endorse (input Alice) to Bob
let valid = declassify (0 ≤ shell ≤ 2) to Bob
if valid:
    let guess = endorse (input Bob) to Alice
    let win = declassify (guess == shell) to Alice ∨ Bob
output win to Alice, Bob
```

Who should execute this?

Alice ?

Bob ?
Synthesis: partitioning the program

host Alice // dealer
host Bob // player

let shell = endorse (input Alice) to Bob
let valid = declassify (0 ≤ shell ≤ 2) to Bob
if valid:
    let guess = endorse (input Bob) to Alice
    let win = declassify (guess == shell) to Alice \(\lor\) Bob
    output win to Alice, Bob

Who should execute this?

MPC
Synthesis: partitioning the program

host Alice    // dealer
host Bob      // player

let shell = endorse (input Alice) to Bob
let valid = \(0 \leq \text{shell} \leq 2\) to Bob
if valid:
  let guess = endorse (input Bob) to Alice
  let win = \(\text{guess} == \text{shell}\) to Alice V Bob
output win to Alice, Bob
Naive synthesis: execute entire program in MPC!

Secure

Inefficient
Avoid using crypto
  e.g. local execution or replicated execution

Use cheaper crypto
  e.g. commitment < ZKP < MPC

... while respecting security
A more efficient synthesis

host Alice // dealer
host Bob // player

let shell = endorse (input Alice) to Bob
let valid = declassify (0 ≤ shell ≤ 2) to Bob
if valid:
  let guess = endorse (input Bob) to Alice
  let win = declassify (guess == shell) to Alice ∨ Bob
output win to Alice, Bob

A performs ZKP
Think of crypto as “principals”

**MPC:** $A \land B$
- neither can see, trusted by A and B

**ZKP:** $A \land B \leftarrow$
- private to A, trusted by A and B
  (by A)

**commit:** $A \land B \leftarrow$
- private to A, trusted by A and B
  (by A)
Lattice defines an ordering ⇒ “acts for” among principals

MPC: $A \land B$

neither can see, trusted by A and B

ZKP : $A \land B$ ←

(by A)

private to A, trusted by A and B

$A$ $B$
Viaduct

Performance profiles
Principal label
Computational capability

Optimized partitioning
Program-to-circuit compiler

Optimized cryptographic implementation

Synthesis pipeline
Check out our open-source implementation

https://viaduct-lang.org
Open questions

Compiler correctness
Open questions

- Compiler correctness
- More expressive performance profiles
  - e.g., bandwidth vs compute
  - boolean vs numeric computation
  - prover vs verifier time
Open questions

- Compiler correctness
- More expressive performance profiles
- Utilize “hand-optimized” capabilities  
  e.g., private set intersection
Open questions

- Compiler correctness
- More expressive performance profiles
- Utilize “hand-optimized” capabilities
- Reason about other security properties e.g., fairness
Thank you!
runting@cs.cmu.edu