Oblivious RAM

Yury Lifshits

Steklov Institute of Mathematics, St.Petersburg, Russia
yura@logic.pdmi.ras.ru

Tartu University
15/03/2006

Architectual Approach to Software Protection

- Computer is divided to observable and protected parts

Architectual Approach to Software Protection

- Computer is divided to observable and protected parts
- Technologically possible: accessable memory but protected processor

Architectual Approach to Software Protection

- Computer is divided to observable and protected parts
- Technologically possible: accessable memory but protected processor
- Today: making interction between processor and memory useless for learning program

Outline

1. What Kind of Computer Are We Going To Construct?

Outline

1. What Kind of Computer Are We Going To Construct?

2. Basic Solutions

Outline

Outline

Computer Model

- Two parts: Memory and Processor

Computer Model

- Two parts: Memory and Processor
- Internal memory of Processor $= c \log |\text{Memory}|$

Computer Model

- Two parts: Memory and Processor
- Internal memory of Processor $= c \log |\text{Memory}|$
- Interaction: fetch(adress), store(adress, value)

Computer Model

- Two parts: Memory and Processor
- Internal memory of Processor $= c \log |\text{Memory}|$
- Interaction: fetch(adress), store(adress, value)
- Processor has access to random oracle

Computer Model

- Two parts: Memory and Processor
- Internal memory of Processor $= c \log |\text{Memory}|$
- Interaction: fetch(adress), store(adress, value)
- Processor has access to random oracle
- Computation starts with a program and an input in Memory

Computer Model

- Two parts: Memory and Processor
- Internal memory of Processor $= c \log |\text{Memory}|$
- Interaction: fetch(adress), store(adress, value)
- Processor has access to random oracle
- Computation starts with a program and an input in Memory
- One step: fetch one cell - update value and Processor memory - store

Oblivious Execution

We want to hide: order of accesses to cells of Memory

Oblivious Execution

We want to hide: order of accesses to cells of Memory

Oblivious esecution:

For all programs of size m working in time t
order of fetch/store adresses is the same

Oblivious Execution

We want to hide: order of accesses to cells of Memory

Oblivious esecution:

For all programs of size m working in time t
order of fetch/store adresses is the same

Weaker requirement:

For all programs of size m working in time t
order of fetch/store adresses has the same distribution

Outline

Naive Simulation

Simulation 1:

1. We store encrypted pairs (adress,value) in memory cells

Naive Simulation

Simulation 1:

1. We store encrypted pairs (adress,value) in memory cells
2. For every fetch/store we scan through all memory

Naive Simulation

Simulation 1:

1. We store encrypted pairs (adress,value) in memory cells
2. For every fetch/store we scan through all memory
    - Wrong adress $\Rightarrow$ just reencrypt and store

Naive Simulation

Simulation 1:

1. We store encrypted pairs (adress,value) in memory cells
2. For every fetch/store we scan through all memory
   - Wrong adress $\Rightarrow$ just reencrypt and store
   - Right adress $\Rightarrow$ do the job $\Rightarrow$ encrypt and store the result

Naive Simulation

Simulation 1:

1. We store encrypted pairs (adress,value) in memory cells
2. For every fetch/store we scan through all memory
   - Wrong adress $\Rightarrow$ just reencrypt and store
   - Right adress $\Rightarrow$ do the job $\Rightarrow$ encrypt and store the result

Naive Simulation

Simulation 1:

1. We store encrypted pairs (adress,value) in memory cells
2. For every fetch/store we scan through all memory
    - Wrong adress ⇒ just reencrypt and store
    - Right adress ⇒ do the job ⇒ encrypt and store the result

Cost of simulation: tm time, m memory

Square Root Solution (1)

We need to protect:

Order of accesses

Number of accesses

Square Root Solution (1)

We need to protect:

Order of accesses

Number of accesses

Memory = Main Part ($m + \sqrt{m}$) | Shelter $\sqrt{m}$

Square Root Solution (1)

We need to protect:

Order of accesses

Number of accesses

Memory = Main Part ($m + \sqrt{m}$) | Shelter $\sqrt{m}$

Idea:

Divide computation in epochs of $\sqrt{m}$ steps each

On each original step make one fetch to the Main Part and scan through all the Shelter

Square Root Solution (2)

Simulation 2:

1. Store input in the Main Part

Square Root Solution (2)

Simulation 2:

1. Store input in the Main Part
2. Add $\sqrt{m}$ dummy cells to the Main part

Square Root Solution (2)

Simulation 2:

1. Store input in the Main Part
2. Add $\sqrt{m}$ dummy cells to the Main part
3. For every epoch of $\sqrt{m}$ steps

Square Root Solution (2)

Simulation 2:

1. Store input in the Main Part
2. Add $\sqrt{m}$ dummy cells to the Main part
3. For every epoch of $\sqrt{m}$ steps
   - Permute all cells in the Main Part (using permutation π from random oracle)

Square Root Solution (2)

Simulation 2:

1. Store input in the Main Part
2. Add $\sqrt{m}$ dummy cells to the Main part
3. For every epoch of $\sqrt{m}$ steps
   - Permute all cells in the Main Part (using permutation π from random oracle)
   - For each process(i) scan through the shelter. If i-th element is not founded, fetch it from $\pi(i)$, otherwise fetch next dummy cell

Square Root Solution (2)

Simulation 2:

1. Store input in the Main Part
2. Add $\sqrt{m}$ dummy cells to the Main part
3. For every epoch of $\sqrt{m}$ steps
    - Permute all cells in the Main Part (using permutation π from random oracle)
    - For each process(i) scan through the shelter. If i-th element is not founded, fetch it from $\pi(i)$, otherwise fetch next dummy cell
    - Update (obliviously) the Main Part using the Shelter values

Square Root Solution (2)

Simulation 2:

1. Store input in the Main Part
2. Add $\sqrt{m}$ dummy cells to the Main part
3. For every epoch of $\sqrt{m}$ steps
   - Permute all cells in the Main Part (using permutation π from random oracle)
   - For each process(i) scan through the shelter. If i-th element is not founded, fetch it from $\pi(i)$, otherwise fetch next dummy cell
   - Update (obliviously) the Main Part using the Shelter values

Square Root Solution (2)

Simulation 2:

1. Store input in the Main Part
2. Add $\sqrt{m}$ dummy cells to the Main part
3. For every epoch of $\sqrt{m}$ steps
   - Permute all cells in the Main Part (using permutation π from random oracle)
   - For each process(i) scan through the shelter. If i-th element is not founded, fetch it from $\pi(i)$, otherwise fetch next dummy cell
   - Update (obliviously) the Main Part using the Shelter values

Cost of simulation: $t\sqrt{m}$ time, $m + 2\sqrt{m}$ memory

Buffer Solution (1): Oblivious Hash Table

Memory of initial program: $(a_1, v_1), \dots, (a_m, v_m)$

- Take a hash function $h : [1..m] \to [1..m]$
- Prepare $m \times \log m$ table
- Put (a_i, v_i) to random free cell in $h(a_i)$-th column
- Home problem 4: Prove that the chance of overflow is less than $1/m$

Buffer Solution (2): Simulation

Restricted problem: assume that every cell accessed only once

Buffer Solution (2): Simulation

Restricted problem: assume that every cell accessed only once

Simulation 3:

1. Construct (obliviously) a hash table

Buffer Solution (2): Simulation

Restricted problem: assume that every cell accessed only once

Simulation 3:

1. Construct (obliviously) a hash table
2. For every step fetch(i) of initial program

Buffer Solution (2): Simulation

Restricted problem: assume that every cell accessed only once

Simulation 3:

1. Construct (obliviously) a hash table
2. For every step fetch(i) of initial program
   - Scan through $h(i)$ column

Buffer Solution (2): Simulation

Restricted problem: assume that every cell accessed only once

Simulation 3:

1. Construct (obliviously) a hash table
2. For every step fetch(i) of initial program
    - Scan through $h(i)$ column
    - Update the target cell

Buffer Solution (2): Simulation

Restricted problem: assume that every cell accessed only once

Simulation 3:

1. Construct (obliviously) a hash table
2. For every step fetch(i) of initial program
   - Scan through $h(i)$ column
   - Update the target cell

Buffer Solution (2): Simulation

Restricted problem: assume that every cell accessed only once

Simulation 3:

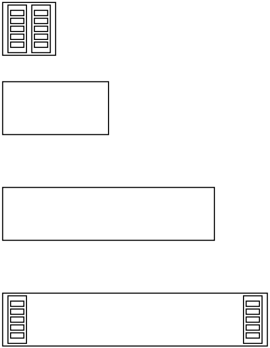
1. Construct (obliviously) a hash table
2. For every step fetch(i) of initial program
   - Scan through $h(i)$ column
   - Update the target cell

Cost of simulation: $t \log m$ time, $m \log m$ memory

Outline

Data Structure

- k-Buffer = table $2^k \times k$
- Hierarchial Buffer Structure = 1-buffer,...,$\log t$-buffer
- Initial position: input in last buffer, all others are empty



Hierarchial Simulation

Simulation of processing cell i:

1. Scan through 1-buffer

Hierarchial Simulation

Simulation of processing cell i:

1. Scan through 1-buffer
2. For every j scan through $h(i,j)$-th column in j-buffer

Hierarchial Simulation

Simulation of processing cell i:

1. Scan through 1-buffer
2. For every j scan through $h(i, j)$-th column in j-buffer
3. Put the updated value to the first buffer

Periodic Rehashing

Refreshing the data structure:

1. Every 2^{j-1} steps unify j-th and $j-1$-th buffers
2. Delete doubles
3. Using new hash function put all data to $j-th$ level

Periodic Rehashing

Refreshing the data structure:

1. Every 2^{j-1} steps unify j-th and $j-1$-th buffers
2. Delete doubles
3. Using new hash function put all data to $j-th$ level

Invariant: For every moment of time for every l buffers from 1 to l all together contain at most 2^{l-1} elements

Discussion

Comments on final solution:

- Cost: $O(t \cdot (\log t)^3)$ time, $O(m \cdot (\log m)^2)$ memory

Discussion

Comments on final solution:

- Cost: $O(t \cdot (\log t)^3)$ time, $O(m \cdot (\log m)^2)$ memory
- Omitted details: realization of oblivious hashing and random oracle

Discussion

Comments on final solution:

- Cost: $O(t \cdot (\log t)^3)$ time, $O(m \cdot (\log m)^2)$ memory
- Omitted details: realization of oblivious hashing and random oracle
- Tamper-proofing extension

Home Problem 4

Prove that the chance of overflow in hash table construction is less than $1/m$

Summary

Main points:

- Theoretical model for hardware-based code protection: open memory/protected CPU

Summary

Main points:

- Theoretical model for hardware-based code protection: open memory/protected CPU
- Central problem: simulation of any program with any input by the same access pattern

Summary

Main points:

- Theoretical model for hardware-based code protection: open memory/protected CPU
- Central problem: simulation of any program with any input by the same access pattern
- Current result: $O(t \cdot (\log t)^3)$ time, $O(m \cdot (\log m)^2)$ memory simulation

Reading List

O. Goldreich, R. Ostrovsky
Software protection and simulation on oblivious RAM, 1996.
http://www.wisdom.weizmann.ac.il/~oded/PS/soft.ps.

Thanks for attention. Questions?