Password Hashing and Memory Hardness

ENEE 457/CMSC 498E

Recall Password Hashing

- Store only the hash values of the passwords in a table in the clear.
  - If Server is compromised, hard to recover password values given hash values.
- To defeat “Rainbow Tables” we can use a salt when hashing the password.

But how to defeat the "Brute Force" Attack?

- Recall, only around $95^6 \approx 7 \times 10^{11}$ hash evaluations required to recover a single password using Brute Force Search.
- Solution:

Password Scrambler PS:

1. Given a password pass, computing $PS(pass)$ should be "fast enough" for the user.
2. Computing $PS(pass)$ should be "as slow as possible" without contradicting 1.
3. Given $y = PS(pass)$ there must be no significantly faster way to test q password candidates than by actually computing PS on each candidate.

What About Parallel Computation?

- Can't a b-core adversary always get a b-times speedup?
- Memory is expensive
  - Typical GPU or other cheap and massively-parallel hardware with lots of cores can only have a limited amount of fast ("cache") memory for each single core
- Make the password scrambler PS not only intentionally slow on standard sequential computers, but also intentionally **memory-consuming**.
- Any adversary using b cores in parallel with less than about b times the memory of a sequential implementation must experience a strong slow-down.

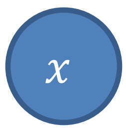
“Memory-Hard Functions”

- Idea:
  - Start with an underlying hash function h
  - Build a bigger hash function H from h

- Assume to compute $h(x_1, \ldots, x_\ell)$ requires ℓ units of time and ℓ units of memory.

Representing Hash Function Evaluation using a Graph

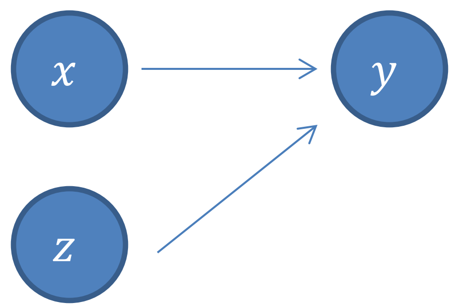
Start node corresponds to input



$y = h(x)$

Each node corresponds to a value stored in memory.

Typically require **in-degree** to be **constant**, so that hash evaluation for each node takes constant time.



$y = h(x||z)$

(Parallel) Graph Pebbling

Let $G = (V, E)$ be a DAG and $T, S \subseteq V$ be node sets. Then a (legal) pebbling of G with starting configuration S and target T is a sequence $(P_0, \ldots, P_t)$of subsets of V such that:

1. $P_0 \subseteq S$
2. Pebbles are added only when their predecessors already have a pebble at the end of the previous step
3. At some point every target node is pebbled (not necessarily simultaneously).

We call a pebbling of G complete if $S = \emptyset$ and T is the set of sink nodes of G.

Space Complexity

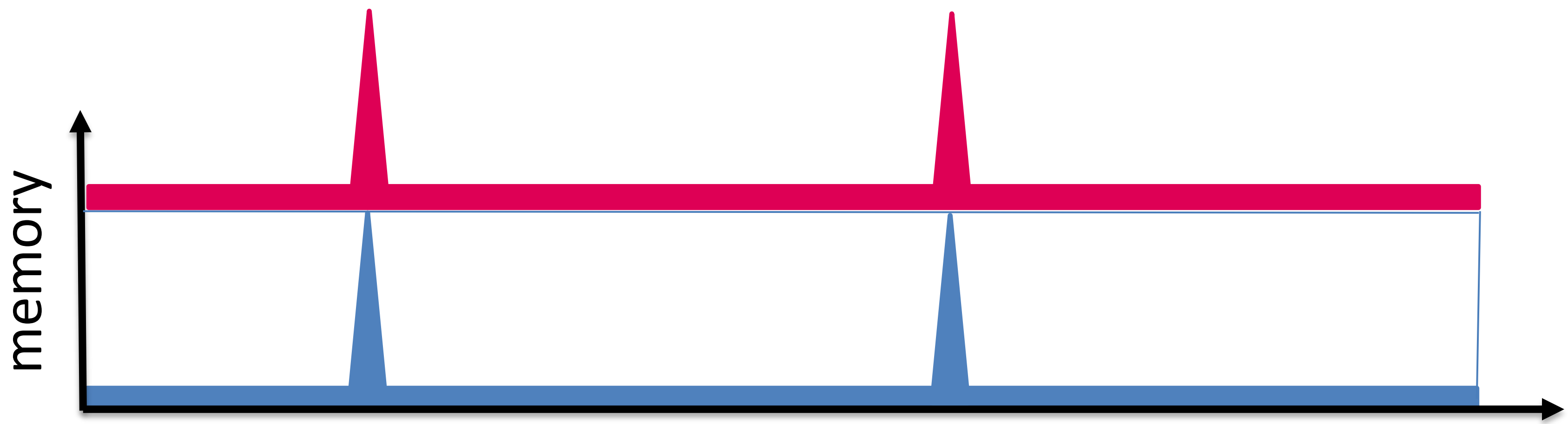
Let G be a DAG, $P = (P_0, \ldots, P_t)$ be an arbitrary pebbling of G and Π be the set of all complete pebblings of G. Then the (cumulative) cost of P and the cumulative complexity (CC) of G are defined respectively to be:

$$s\text{-}cost(P) \coloneqq \max\{P_i : i \in \{0, \ldots, t\}\}$$
$$sc(G) \coloneqq \min\left\{s\text{-}cost(P) : P \in \Pi\right\}$$

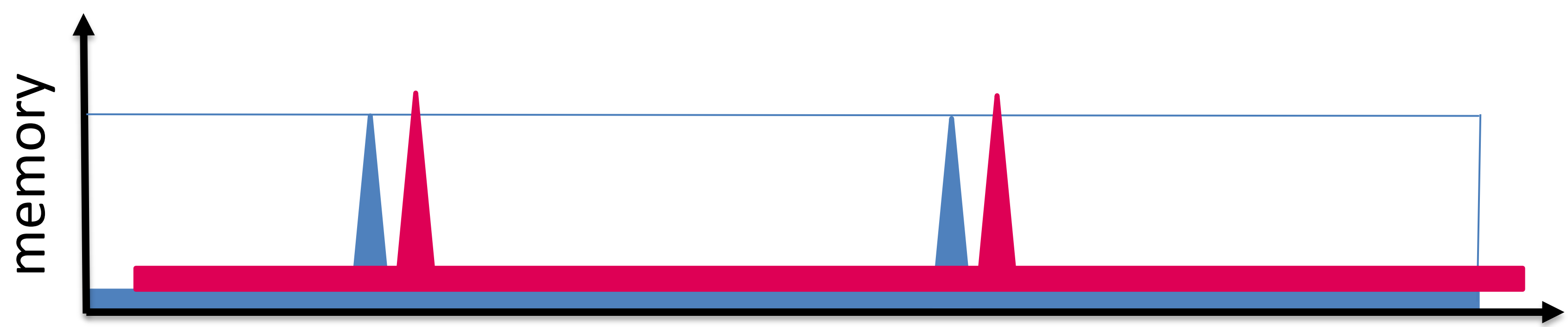
$$st{-}cost(P) \coloneqq t \cdot \max\{P_i : i \in \{0, \ldots, t\}\}$$
$$stc(G) \coloneqq \min\{st{-}cost(P) : P \in \Pi\}$$

Problem with Standard Notions

To compute two instances, a smart adversary won't do this!

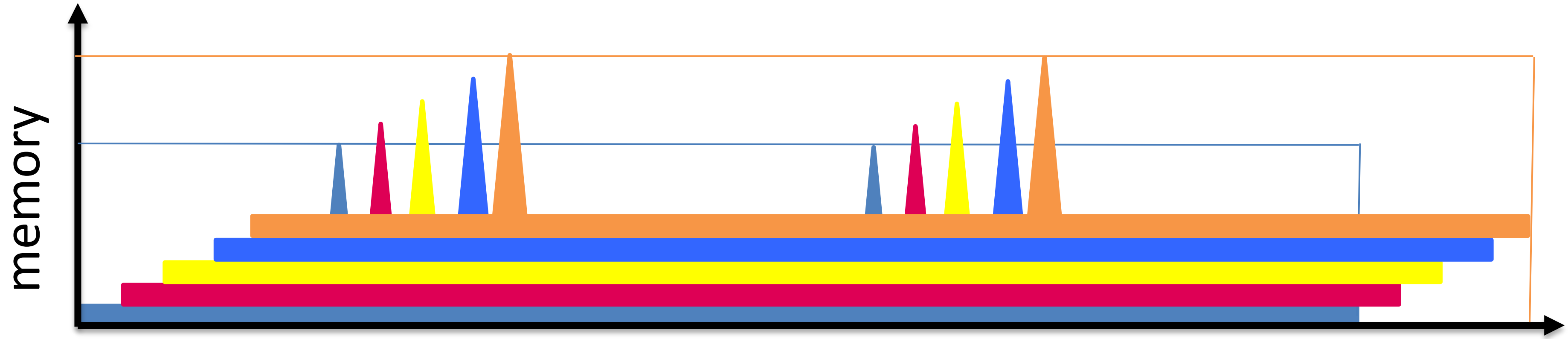


Instead:



Problem with Standard Notions

Offset multiple computations by a little to keep cost low!



Cumulative Pebbling Complexity

Let G be a DAG, $P = (P_0, \ldots, P_t)$ be an arbitrary pebbling of G and Π be the set of all complete pebblings of G. Then the (cumulative) cost of P and the cumulative complexity (CC) of G are defined respectively to be:

$$p\text{-}cost(P) \coloneqq \sum_{i=0}^{t} |P_i|$$

$$cc(G) \coloneqq \min\{p\text{-}cost(P) : P \in \Pi\}$$

Cumulative Pebbling Complexity

Lemma: Let $G = G_1 + G_2$. Then $cc(G) = cc(G_1) + cc(G_2)$.

Lemma: There exists a G such that $stc(G^{\times n}) = O\left(stc(G)\right)$.

Maximal CC?

Lemma: Let G be a DAG of size n and depth d. Then $cc(G) \leq dn$.

Maximal CC is at most n^2 for an n-node graph.

Our focus: What is the Maximal CC we can achieve for graphs with constant in-degree?

Case Study: Bit-Reversal Graphs

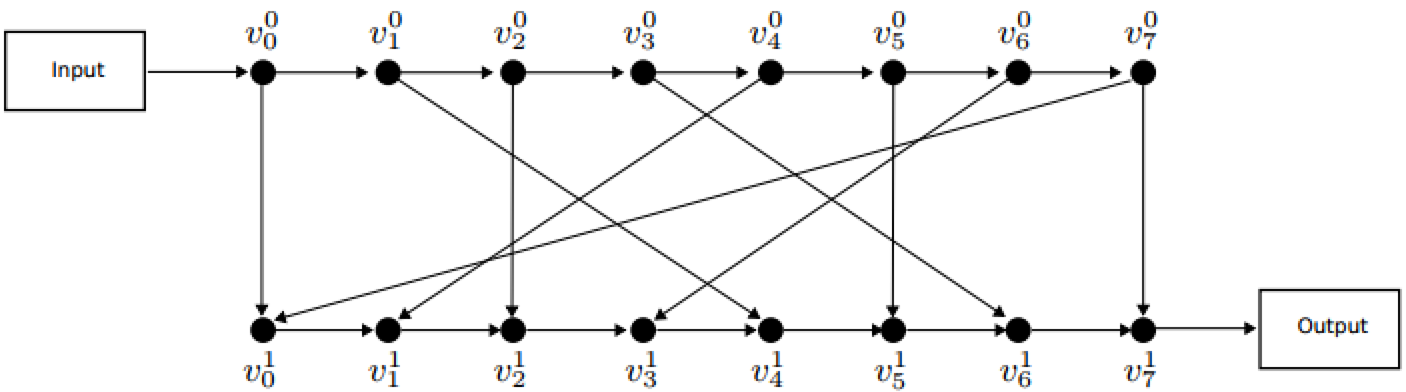


Fig. 1. An $(8, 1)$-BRG.

Case Study: Bit-Reversal Graphs

Algorithm 2 (g, λ)-Bit-Reversal Hashing (BRH_λ^g)

Require: g {Garlic}, x {Value to Hash}, λ {Depth}, H {Hash Function}
Ensure: x {Password Hash}

$v_0 \leftarrow H(x)$
for $i = 1, \ldots, 2^g - 1$ **do**
    $v_i \leftarrow H(v_{i-1})$
end for
for $k = 1, \ldots, \lambda$ **do**
    $r_0 \leftarrow H(v_0 \,||\, v_{2^g-1})$
    for $i = 1, \ldots, 2^g - 1$ **do**
        $r_i \leftarrow H(r_{i-1} \,||\, v_{\tau(i)})$
    end for
    $v \leftarrow r$
end for
return r_{2^g-1}

Case Study: Bit-Reversal Graphs

It was shown in [Lengauer Tarjan 82] that (in the sequential setting) any pebbling using S pebbles requires time T such that ST = $O(n^2)$.

Such graphs were suggested as candidates for password hashing. E.g. in the **Catena** framework (finalist in Password Hashing Competition [PHC]).

We will describe an algorithm which can pebble the bit-reversal graph of size n using cumulative cost of at most $O(n^{1.5})$.

CC of Bit-Reversal Graphs?

Theorem: A Bit-Reversal graph G of size n has $cc(G) = O(n^{1.5})$.

Extends to any "sandwich" graph:

A chain of n nodes (numbered 1 through n) with arbitrary additional edges connecting nodes from the first half of the chain with nodes of the second half of the chain such that no node has in-degree greater than 2.

CC of Bit-Reversal Graphs?

Proof: Consider the following strategy:

1. If $i \ mod \ \sqrt{n} = 0$ then place a pebble on node 1
2. For each pebble on a node $v \in [n]$ place a pebble on node $v + 1$
3. Remove any pebble on nodes $\left\{\left(\frac{n}{2}\right) + 1, \dots, n\right\}$ except the one on the highest valued node.
4. Let m be the highest valued node with a pebble on it. Remove any pebbles on nodes $v \in [n/2]$ except if $(i - v) mod \ \sqrt{n} = 0$ or if there is an edge $(v, m + j)$ for some $0 < j < \sqrt{n}$ and $m + j > n/2$.

CC of Bit-Reversal Graphs?

Proof (cont'd).

Must show (1) the above strategy is a legal pebbling (2) at any time there are $O(\sqrt{n})$ pebbles on the graph.

For (1), must show that this step is legal: For each pebble on a node $v \in [n]$ place a pebble on node $v+1$. Legal for first n/2 nodes, but not necessarily second n/2 nodes. Why?

Key: for second n/2 nodes only the highest pebble on node m remains from previous round (due to Rule 3). Node m+1 has at most one additional incoming edge from first n/2 nodes. This node must be pebbled due to the fact that each node is touched every $\sqrt{n}$ iterations and the second half of Rule 4.

For (2), at most one pebble on second n/2 nodes (due to Rule 3). Due to first half of Rule 4, at most $\sqrt{n}$ pebbles remain. Due to second half of Rule 4 and the fact that each of the second n/2 nodes has in-degree at most 2, at most an additional $2\sqrt{n}$ pebbles remain.

Scrypt

- Initially introduced by Percival ‘09.
- Used in proofs-of-work schemes for cryptocurrencies.
- Inspired the design of one of the Password-hashing Competition’s [PHC] winners, Argon2d.
- Similar structure to “sandwich” graph, but is data-dependent.
- The edges in the graph depend on the outcome of the hashed data.

Scrypt

- Input X
- Output S_n
- $X_0 = X$ and for $i = 1, \ldots, n-1$: $X_i = h(X_{i-1})$
- $S_0 = h\left(X_{n-1)}\right)$ and for $i = 1, \ldots, n$: $S_i = h\left(S_{i-1} \oplus X_{S_{i-1}\, mod\, n}\right)$

Scrypt is Maximally Memory Hard

Theorem (Alwen et al.): The cumulative complexity of Scrypt is $\Omega(n^2)$.