Targeted Pseudorandom Generators, Simulation Advice Generators, and Derandomizing Logspace

William M. Hoza (UT Austin) and **Chris Umans** (Caltech)

Derandomization vs. Pseudorandom Generators

- PRG $\implies$ derandomization. What about the other way?
- Best PRG for **BPL**: Nisan '92: Seed length $O(\log^2 n)$
- Best derandomization: Saks, Zhou '99: $\mathbf{BPL} \subseteq \mathbf{DSPACE}(\log^{3/2} n)$
- **Theorem** (Main result, simplest version):
  - Assume that for every derandomization result for logspace algorithms, there is a PRG strong enough to (nearly) recover derandomization by iterating over all seeds and taking a majority vote
  - Then $\mathbf{BPL} \subseteq \bigcap_{\alpha>0} \mathbf{DSPACE}(\log^{1+\alpha} n)$

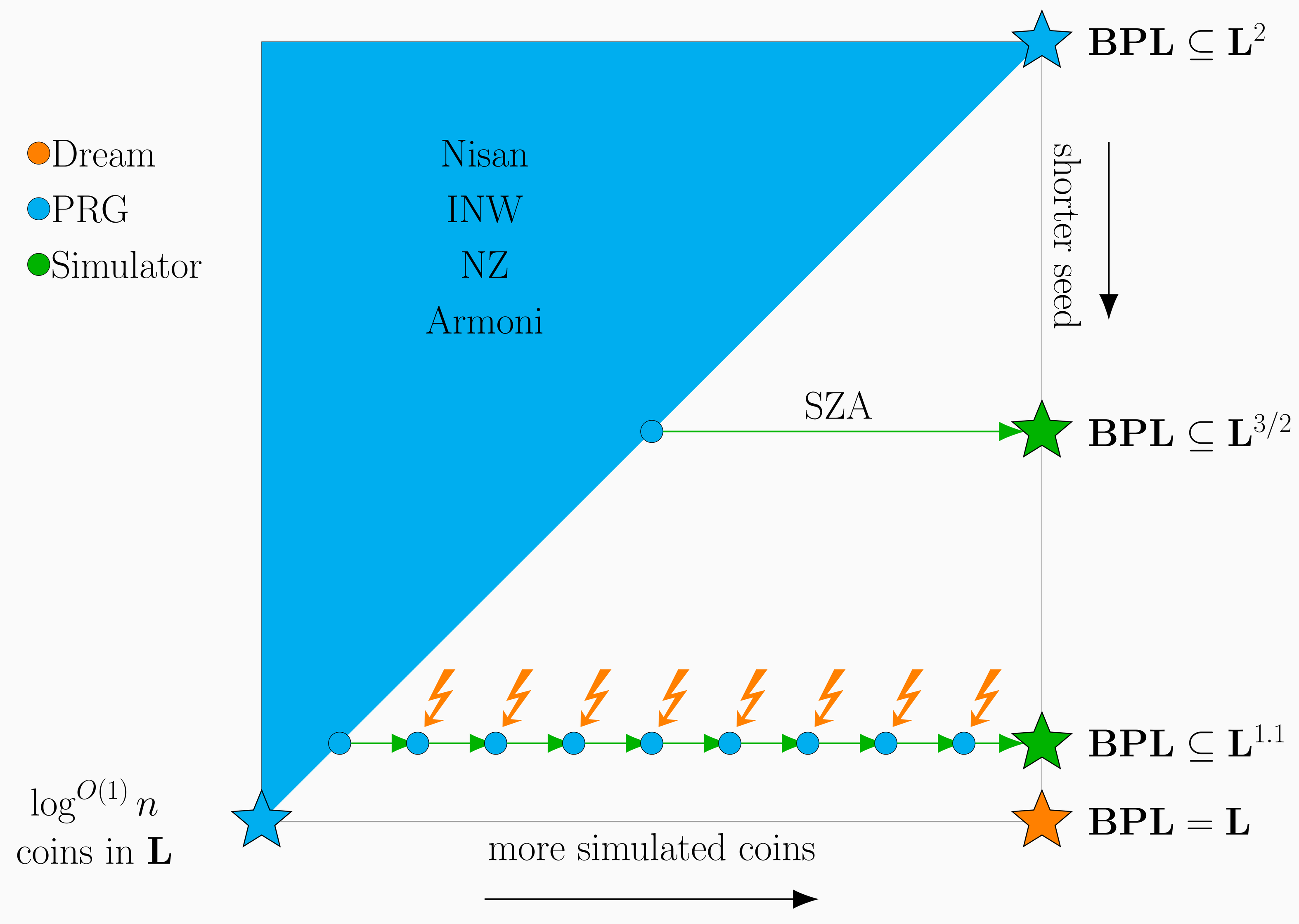
Randomness-efficient simulators for automata

- Nonuniform model of $\log n$ space: n-state automaton
- $Q^m(q; y) \stackrel{\text{def}}{=}$ final state if Q starts in state q, reads $y \in \{0,1\}^m$
- Simulator: Algorithm Sim such that $\mathsf{Sim}(Q, q, U_s) \sim_\epsilon Q^m(q; U_m)$
  - Generic derandomizer, good enough for $\mathbf{L} = \mathbf{BPL}$
- In contrast, a PRG doesn't see "source code" (Q, q) – bonus feature!
- Assumption of main result: For every simulator, there is a PRG with similar parameters

Main tool: Saks-Zhou-Armoni transformation

- What do you do when your PRG doesn't output enough bits?
- Assume oracle access to $\mathsf{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for n-state automata
- Could we use Gen as subroutine in new PRG?
  - INW '94: To get m pseudorandom bits, use seed length
$$s + O\left(\log n \cdot \log\left(\frac{m}{m_0}\right)\right)$$
- **Theorem** (implicit in Armoni '98, builds on Saks, Zhou '99):
  - Given oracle Gen, can construct m-step simulator for n-state automata with seed length/space complexity
$$O\left(s + (\log n) \cdot \frac{\log m}{\log m_0}\right)$$
- Example: To recover Saks-Zhou theorem, let Gen be the INW generator with $m_0 = 2^{\sqrt{\log n}}$, $s = O(\log^{3/2} n)$, $m = n$

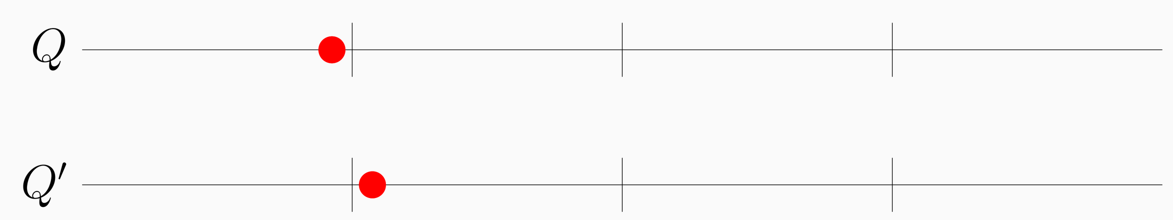
Proof of main result



- Scaling:
  - Seed length is $\log^{1+y} n$, with $0 \le y \le 1$
  - # simulated coins is $2^{\log^x n}$, with $0 \le x \le 1$

Proof idea of SZA theorem

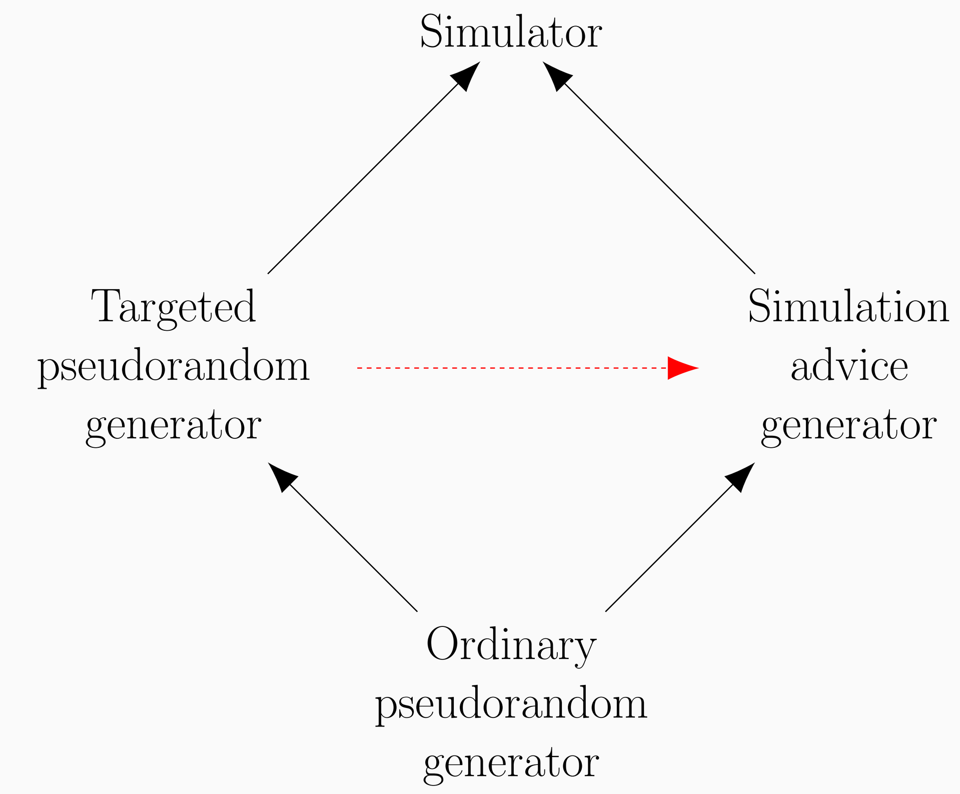
- $O(s)$-coin subroutine Pow: Given automaton Q, produce automaton $\mathsf{Pow}(Q) \approx Q^{m_0}$
  - Let $\mathsf{Samp} : \{0,1\}^{O(s)} \times \{0,1\}^{O(\log n)} \to \{0,1\}^s$ be an averaging sampler
  - For an automaton Q, let $\mathsf{Pow}(Q, x)$ be the automaton defined by
$$\mathsf{Pow}(Q, x)(q; y) = Q^{m_0}(q; \mathsf{Gen}(\mathsf{Samp}(x, y)))$$
  - With high probability over x, $\mathsf{Pow}(Q, x) \approx Q^{m_0}$
  - Note that $\mathsf{Pow}(Q, x)$ reads $O(\log n)$ bits at a time
- Could we just compute $\mathsf{Pow}(\mathsf{Pow}(\mathsf{Pow}(\cdots(\mathsf{Pow}(Q))\cdots)))$ to approximate Q^m?
  - Total # coins $O(s \cdot \frac{\log m}{\log m_0})$. Too many
- Therefore, reuse randomness of Pow in each iteration
  - Difficulty: $\mathsf{Pow}(Q, x)$ is stochastically dependent on x, so why should $\mathsf{Pow}(\mathsf{Pow}(Q, x), x)$ have low failure probability?
- Key: to break stochastic dependencies, perturb and round automaton after each Pow



- With high probability, after perturbing and rounding, arrive at automaton we would have reached with exact powering

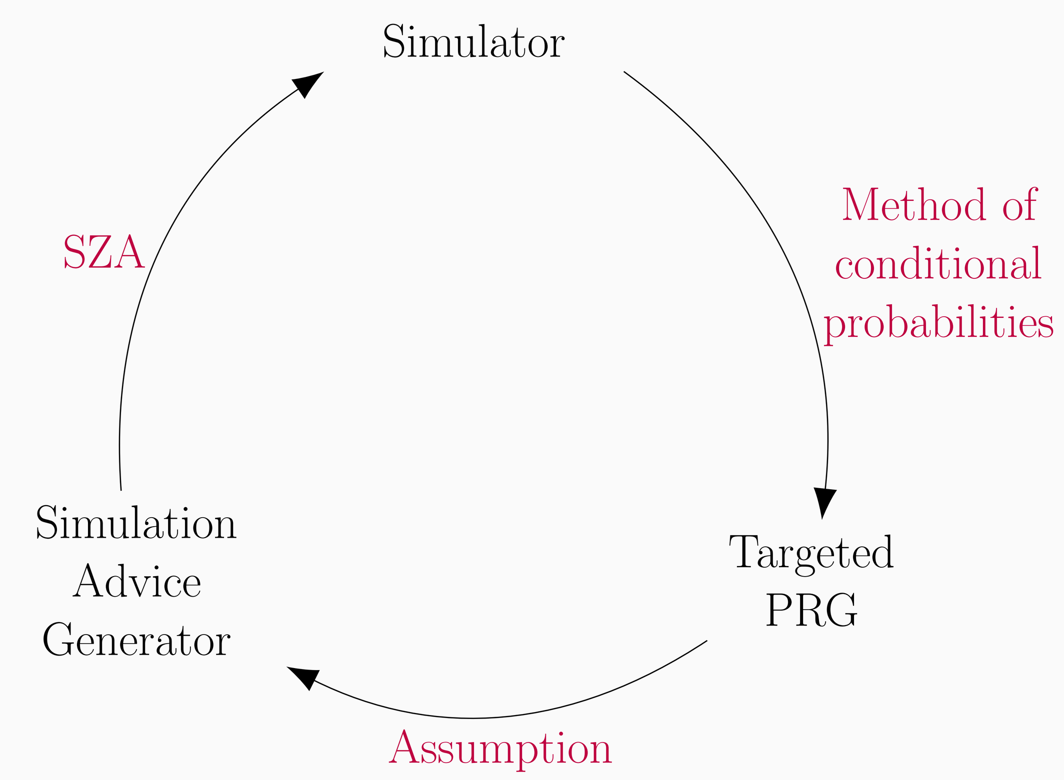
Four kinds of derandomization

- Targeted PRG:
  - Inputs: Automaton Q, start state q, seed $x \in \{0,1\}^s$
  - Output: Bitstring $y \in \{0,1\}^m$ that looks random to $Q^m(q; \cdot)$
- Simulation advice generator:
  - Input: Seed $x \in \{0,1\}^s$
  - Output: Advice $y \in \{0,1\}^a$ such that $Q^m(q; U_m)$ can be simulated in logspace given Q, q, y



Main result, strong version

- **Theorem**: The following are equivalent:
  1. For every targeted pseudorandom generator, there is a simulation advice generator with similar parameters
  2. $\bigcap_{\alpha>0}$ **promise-BPSPACE**$(\log^{1+\alpha} n) = \bigcap_{\alpha>0}$ **promise-DSPACE**$(\log^{1+\alpha} n)$
- Proof idea:



This material is based upon work supported by NSF GRFP Grant No. DGE-1610403 and NSF Grant No. NSF CCF-1423544.