Pseudorandom Generators vs. Derandomization for Logspace Algorithms

(Paper title: "Targeted Pseudorandom Generators, Simulation Advice Generators, and Derandomizing Logspace")

William M. Hoza[1] Chris Umans[2]

June 21, 2017
STOC

[1]University of Texas at Austin
[2]California Institute of Technology

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

    $$\mathbf{AM} \subseteq$$

  - Then there is a PRG that gives that same derandomization

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

$$\mathbf{AM} \subseteq \quad \mathbf{\Sigma_2}$$

  - Then there is a PRG that gives that same derandomization

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

$$\mathbf{AM} \subseteq \bigcap_{\varepsilon>0} \mathbf{\Sigma_2 TIME}(2^{n^{\varepsilon}})$$

  - Then there is a PRG that gives that same derandomization

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

$$\textbf{promise-AM} \subseteq \bigcap_{\varepsilon>0} \mathbf{\Sigma_2 TIME}(2^{n^{\varepsilon}})$$

  - Then there is a PRG that gives that same derandomization

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

$$\textbf{promise-AM} \subseteq \bigcap_{\varepsilon>0} \textbf{i.o.-}\boldsymbol{\Sigma_2}\textbf{TIME}(2^{n^{\varepsilon}})$$

  - Then there is a PRG that gives that same derandomization

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

$$\textbf{promise-AM} \subseteq \bigcap_{\varepsilon>0} \textbf{i.o.-}\boldsymbol{\Sigma}_{\mathbf{2}}\textbf{TIME}(2^{n^{\varepsilon}})/n^{\varepsilon}$$

  - Then there is a PRG that gives that same derandomization

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

$$\textbf{promise-AM} \subseteq \bigcap_{\varepsilon>0} \textbf{i.o.-}\boldsymbol{\Sigma}_\mathbf{2}\textbf{TIME}(2^{n^\varepsilon})/n^\varepsilon$$

  - Then there is a PRG that gives that same derandomization
- Theorem (Goldreich '11):

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

$$\textbf{promise-AM} \subseteq \bigcap_{\varepsilon>0} \textbf{i.o.-}\boldsymbol{\Sigma}_2\textbf{TIME}(2^{n^{\varepsilon}})/n^{\varepsilon}$$

  - Then there is a PRG that gives that same derandomization
- Theorem (Goldreich '11):
  - Assume that for every problem in **promise-BPP**, there is a deterministic polytime algorithm that succeeds on all feasibly generated inputs

Derandomization $\overset{?}{\Longleftrightarrow}$ PRG

- Theorem (Aydınlıoğlu, van Melkebeek '12):
  - Assume the following derandomization statement:

$$\textbf{promise-AM} \subseteq \bigcap_{\varepsilon > 0} \textbf{i.o.-}\mathbf{\Sigma_2}\textbf{TIME}(2^{n^{\varepsilon}})/n^{\varepsilon}$$

  - Then there is a PRG that gives that same derandomization
- Theorem (Goldreich '11):
  - Assume that for every problem in **promise-BPP**, there is a deterministic polytime algorithm that succeeds on all feasibly generated inputs
  - Then there is a PRG that gives that same derandomization

L vs. BPL

L vs. BPL

- Best PRG against logspace (Nisan '92): Seed length

$$O(\log^2 n)$$

L vs. BPL

- Best PRG against logspace (Nisan '92): Seed length

$$O(\log^2 n)$$

- Best derandomization (Saks, Zhou '99):

$$\mathbf{BPL} \subseteq \mathbf{DSPACE}(\log^{3/2} n)$$

Simplest version of main result

- **Theorem** (informally stated):

Simplest version of main result

- **Theorem** (informally stated):
  - Assume that for every derandomization result for logspace algorithms, there is a PRG strong enough to (nearly) recover derandomization by iterating over all seeds

Simplest version of main result

- **Theorem** (informally stated):
  - Assume that for every derandomization result for logspace algorithms, there is a PRG strong enough to (nearly) recover derandomization by iterating over all seeds
  - Then

$$\mathbf{BPL} \subseteq \bigcap_{\alpha > 0} \mathbf{DSPACE}(\log^{1+\alpha} n).$$

Simplest version of main result

- **Theorem** (informally stated):
  - Assume that for every derandomization result for logspace algorithms, there is a PRG strong enough to (nearly) recover derandomization by iterating over all seeds
  - Then

$$\mathbf{BPL} \subseteq \bigcap_{\alpha > 0} \mathbf{DSPACE}(\log^{1+\alpha} n).$$

- Equivalence of PRGs and derandomization would itself give a derandomization!

How to interpret our result

How to interpret our result

How to interpret our result

When your PRG doesn't output enough bits

When your PRG doesn't output enough bits

- Given: Oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $\log n$ space

When your PRG doesn't output enough bits

- Given: Oracle $\text{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $\log n$ space
- Goal: Simulate $(\log n)$-space m-coin algorithm, $m \gg m_0$

When your PRG doesn't output enough bits

- Given: Oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $\log n$ space
- Goal: Simulate $(\log n)$-space m-coin algorithm, $m \gg m_0$
- Approach 1: Ignore oracle, use a PRG which outputs m bits

When your PRG doesn't output enough bits

- Given: Oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $\log n$ space
- Goal: Simulate $(\log n)$-space m-coin algorithm, $m \gg m_0$
- Approach 1: Ignore oracle, use a PRG which outputs m bits
  - E.g. INW '94 (extractors): Seed length $O(\log n \log m)$

When your PRG doesn't output enough bits

- Given: Oracle $\text{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $\log n$ space
- Goal: Simulate $(\log n)$-space m-coin algorithm, $m \gg m_0$
- Approach 1: Ignore oracle, use a PRG which outputs m bits
  - E.g. INW '94 (extractors): Seed length $O(\log n \log m)$
- Approach 2: Use Gen as building block in new PRG which outputs m bits

When your PRG doesn't output enough bits

- Given: Oracle Gen : $\{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $\log n$ space
- Goal: Simulate $(\log n)$-space m-coin algorithm, $m \gg m_0$
- Approach 1: Ignore oracle, use a PRG which outputs m bits
  - E.g. INW '94 (extractors): Seed length $O(\log n \log m)$
- Approach 2: Use Gen as building block in new PRG which outputs m bits
  - E.g. using techniques of INW: Seed length

$$s + O\left((\log n) \cdot \log\left(\frac{m}{m_0}\right)\right)$$

When your PRG doesn't output enough bits

- Given: Oracle Gen : $\{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $\log n$ space
- Goal: Simulate $(\log n)$-space m-coin algorithm, $m \gg m_0$
- Approach 1: Ignore oracle, use a PRG which outputs m bits
  - E.g. INW '94 (extractors): Seed length $O(\log n \log m)$
- Approach 2: Use Gen as building block in new PRG which outputs m bits
  - E.g. using techniques of INW: Seed length

$$s + O\left((\log n) \cdot \log\left(\frac{m}{m_0}\right)\right)$$

  - For $m \gg m_0$, might as well have started from scratch!

When your PRG doesn't output enough bits

- Given: Oracle $\text{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $\log n$ space
- Goal: Simulate $(\log n)$-space m-coin algorithm, $m \gg m_0$
- Approach 1: Ignore oracle, use a PRG which outputs m bits
  - E.g. INW '94 (extractors): Seed length $O(\log n \log m)$
- Approach 2: Use Gen as building block in new PRG which outputs m bits
  - E.g. using techniques of INW: Seed length

$$s + O\left((\log n) \cdot \log\left(\frac{m}{m_0}\right)\right)$$

  - For $m \gg m_0$, might as well have started from scratch!
- Approach 3: Use Gen as building block in simulator

Randomness-efficient simulator

- Inputs:

Randomness-efficient simulator

- Inputs:
  - "Source code" of $(\log n)$-space, m-coin algorithm A

Randomness-efficient simulator

- Inputs:
  - “Source code” of $(\log n)$-space, m-coin algorithm A
  - s-bit seed

Randomness-efficient simulator

- Inputs:
  - "Source code" of $(\log n)$-space, m-coin algorithm A
  - s-bit seed
- (Output of simulator) $\sim_{\varepsilon}$ (final configuration of A)

Randomness-efficient simulator

- Inputs:
  - "Source code" of $(\log n)$-space, m-coin algorithm A
  - s-bit seed
- (Output of simulator) $\sim_{\varepsilon}$ (final configuration of A)
- A PRG induces a simulator

Randomness-efficient simulator

- Inputs:
  - "Source code" of $(\log n)$-space, m-coin algorithm A
  - s-bit seed
- (Output of simulator) $\sim_\varepsilon$ (final configuration of A)
- A PRG induces a simulator
- Crucial bonus feature: PRG doesn't see "source code"!

Saks-Zhou-Armoni transformation

- **Theorem** (implicit in Armoni '98, builds on SZ '99, some details suppressed):

Saks-Zhou-Armoni transformation

- **Theorem** (implicit in Armoni '98, builds on SZ '99, some details suppressed):
  - Given oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $(\log n)$-space algorithms

Saks-Zhou-Armoni transformation

- **Theorem** (implicit in Armoni '98, builds on SZ '99, some details suppressed):
  - Given oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $(\log n)$-space algorithms
  - Can construct simulator for $(\log n)$-space m-coin algorithms with seed length/space complexity

$$O\left(s + (\log n) \cdot \frac{\log m}{\log m_0}\right)$$

Saks-Zhou-Armoni transformation

- **Theorem** (implicit in Armoni '98, builds on SZ '99, some details suppressed):
  - Given oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $(\log n)$-space algorithms
  - Can construct simulator for $(\log n)$-space m-coin algorithms with seed length/space complexity

$$O\left(s + (\log n) \cdot \frac{\log m}{\log m_0}\right)$$

- Original application: Saks-Zhou theorem

Saks-Zhou-Armoni transformation

- **Theorem** (implicit in Armoni '98, builds on SZ '99, some details suppressed):
  - Given oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $(\log n)$-space algorithms
  - Can construct simulator for $(\log n)$-space m-coin algorithms with seed length/space complexity

$$O\left(s + (\log n) \cdot \frac{\log m}{\log m_0}\right)$$

- Original application: Saks-Zhou theorem
  - $m_0 = 2^{\sqrt{\log n}}$, $s = O(\log n \log m_0) = O(\log^{3/2} n)$ (INW)

Saks-Zhou-Armoni transformation

- **Theorem** (implicit in Armoni '98, builds on SZ '99, some details suppressed):
  - Given oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $(\log n)$-space algorithms
  - Can construct simulator for $(\log n)$-space m-coin algorithms with seed length/space complexity

$$O\left(s + (\log n) \cdot \frac{\log m}{\log m_0}\right)$$

- Original application: Saks-Zhou theorem
  - $m_0 = 2^{\sqrt{\log n}}$, $s = O(\log n \log m_0) = O(\log^{3/2} n)$ (INW)
  - Pick $m = n$ (max possible # coins)

Saks-Zhou-Armoni transformation

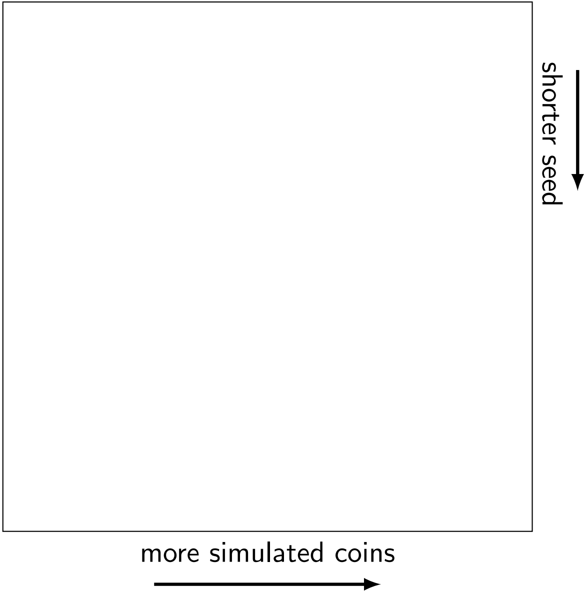
- **Theorem** (implicit in Armoni '98, builds on SZ '99, some details suppressed):
  - Given oracle $\mathrm{Gen} : \{0,1\}^s \to \{0,1\}^{m_0}$, a PRG for $(\log n)$-space algorithms
  - Can construct simulator for $(\log n)$-space m-coin algorithms with seed length/space complexity

$$O\left(s + (\log n) \cdot \frac{\log m}{\log m_0}\right)$$

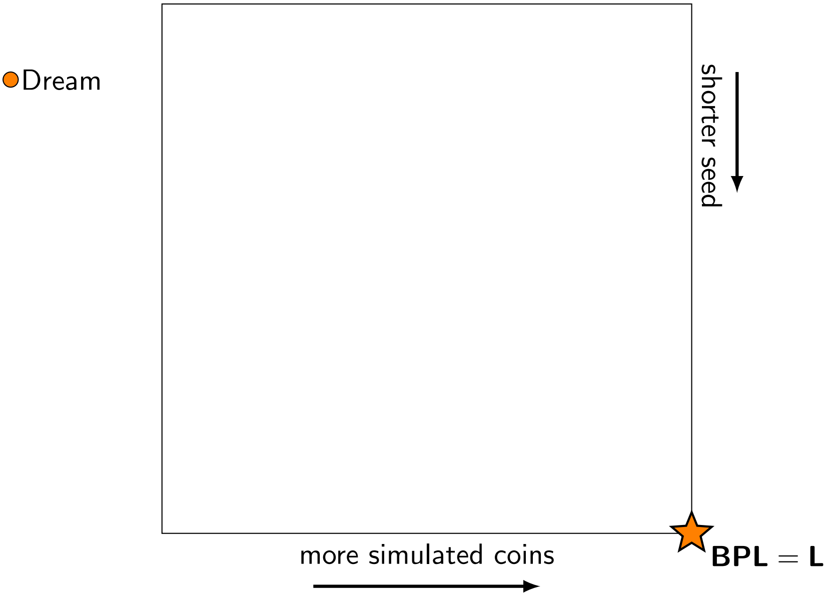
- Original application: Saks-Zhou theorem
  - $m_0 = 2^{\sqrt{\log n}}$, $s = O(\log n \log m_0) = O(\log^{3/2} n)$ (INW)
  - Pick $m = n$ (max possible # coins)
  - $\implies$ simulator with seed length/space complexity

$$O(\log^{3/2} n + \log^{3/2} n) = O(\log^{3/2} n)$$

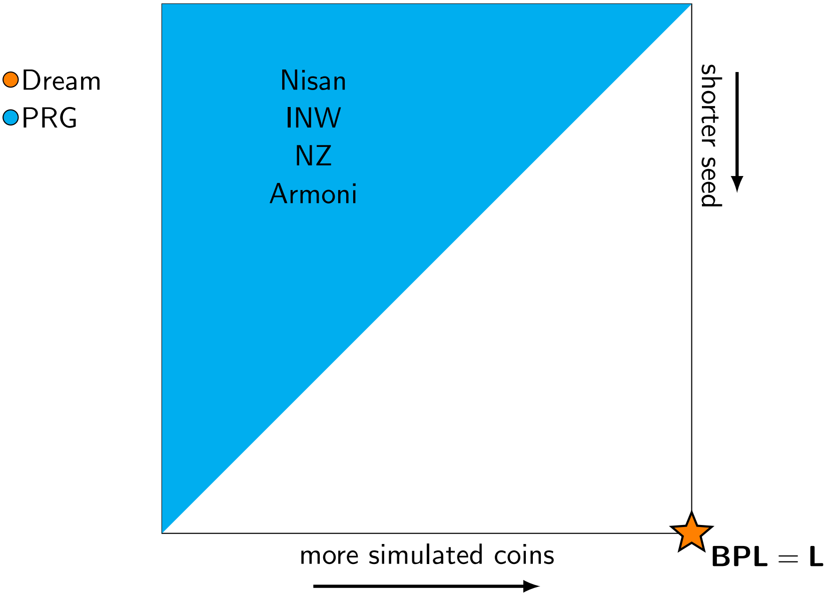
Proof of main result



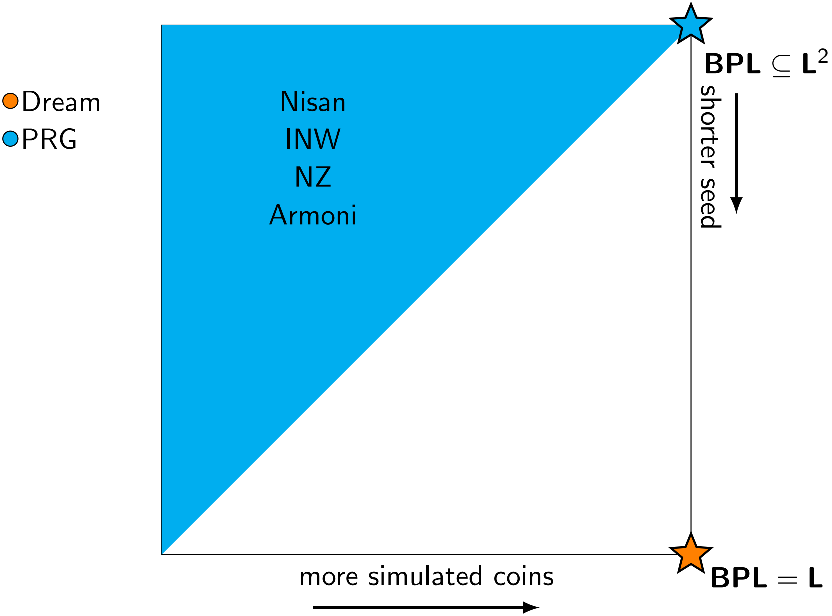
Proof of main result



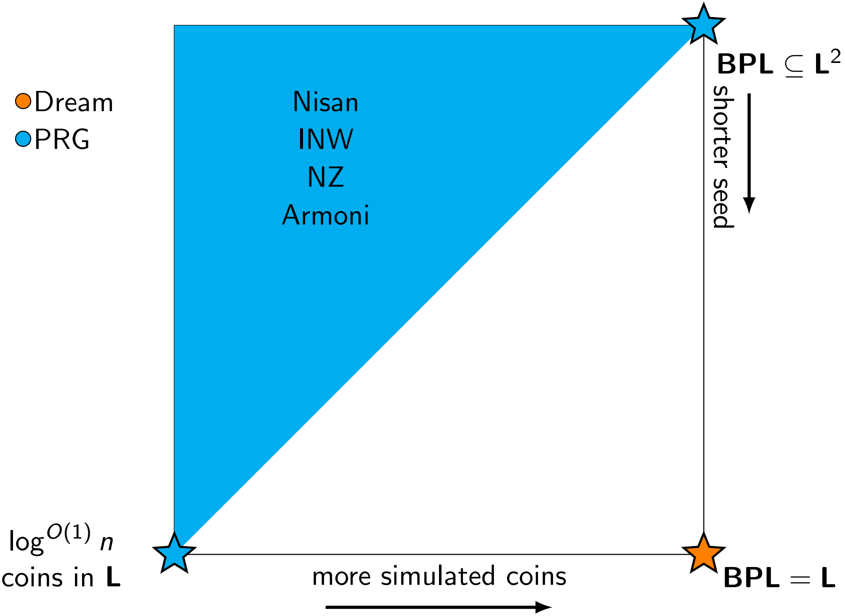
Proof of main result



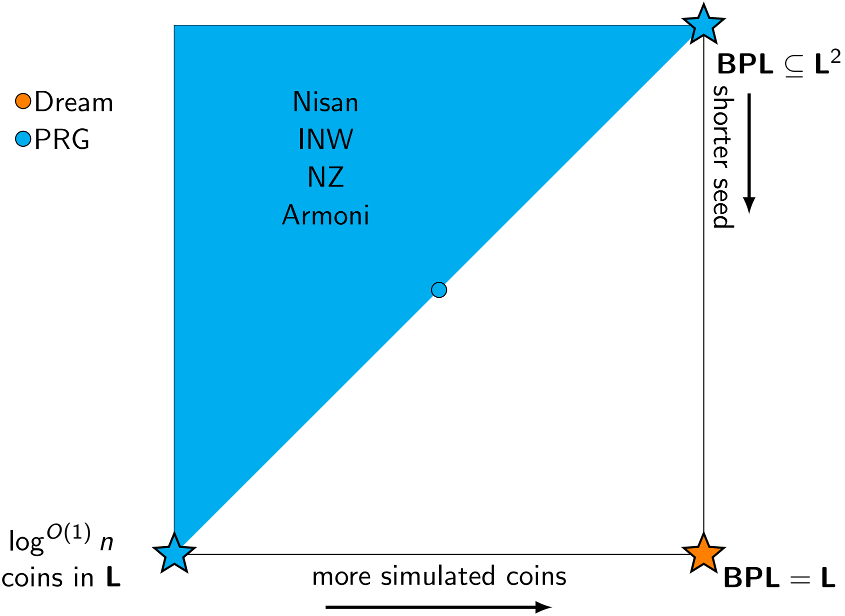
Proof of main result



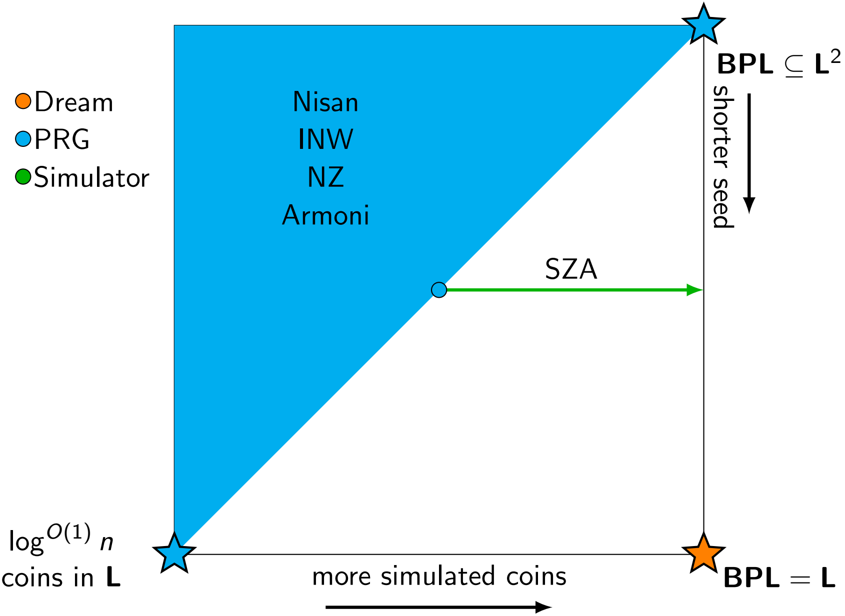
Proof of main result



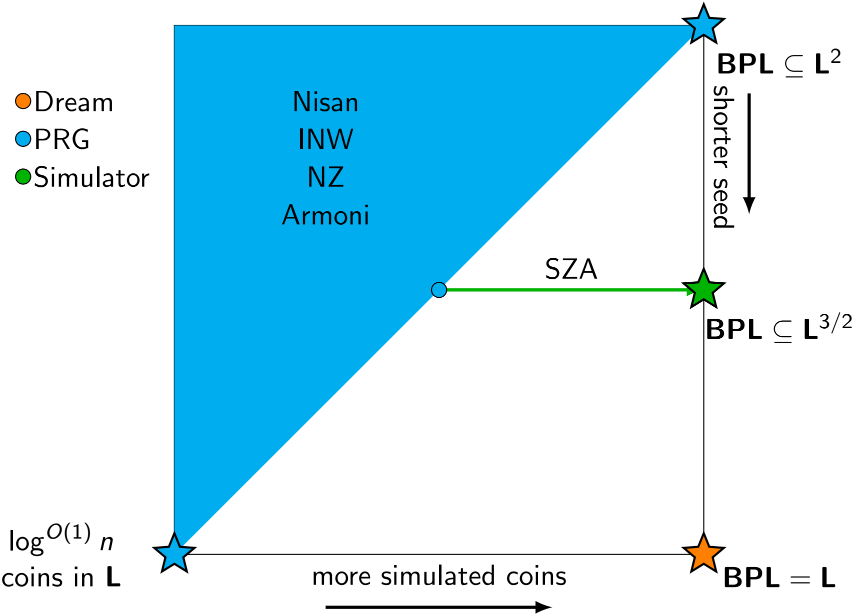
Proof of main result



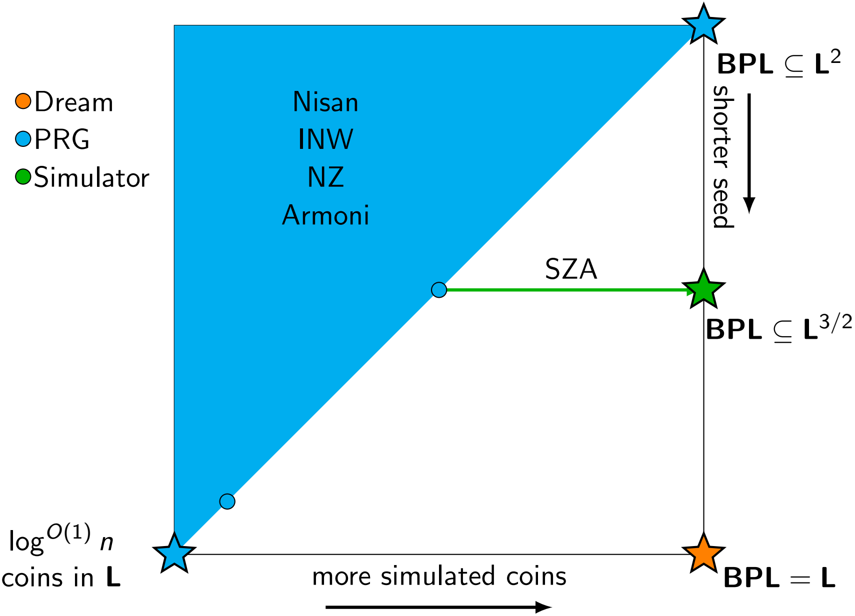
Proof of main result



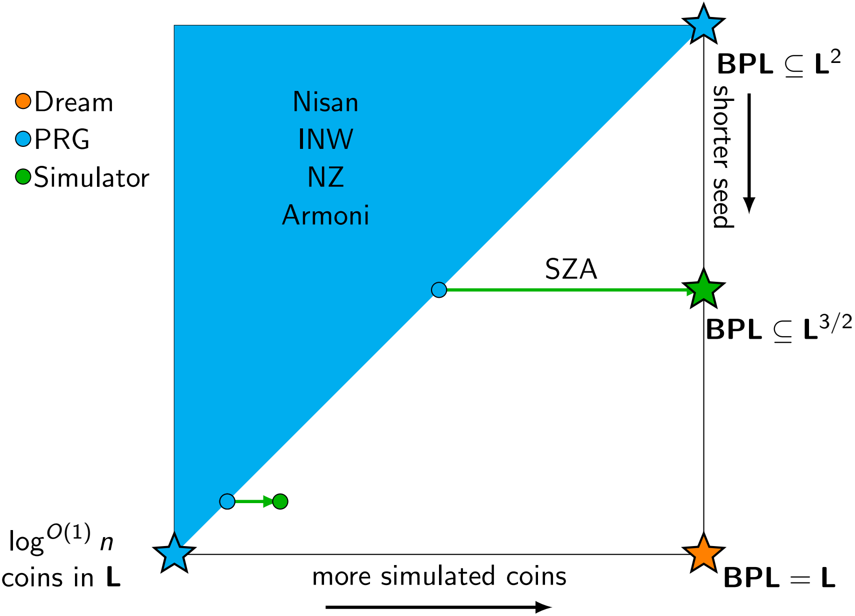
Proof of main result



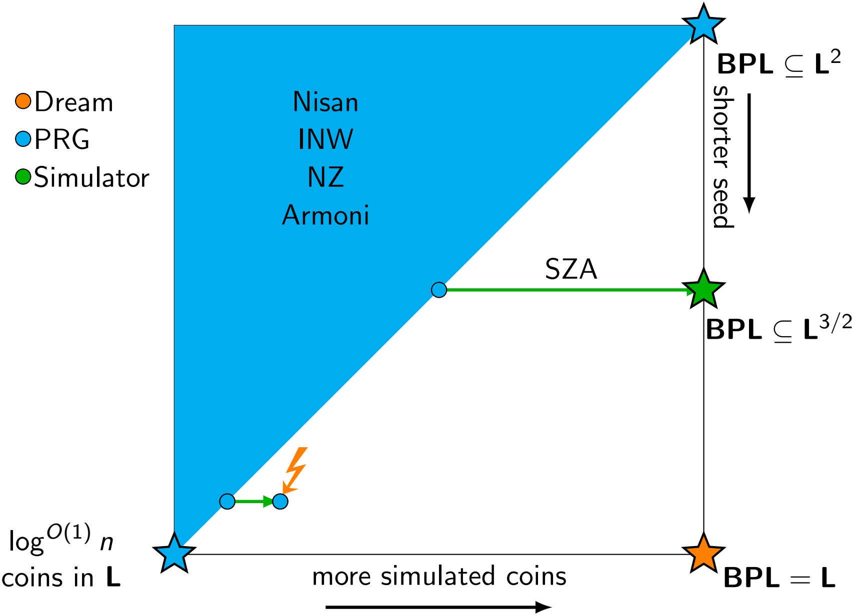
Proof of main result



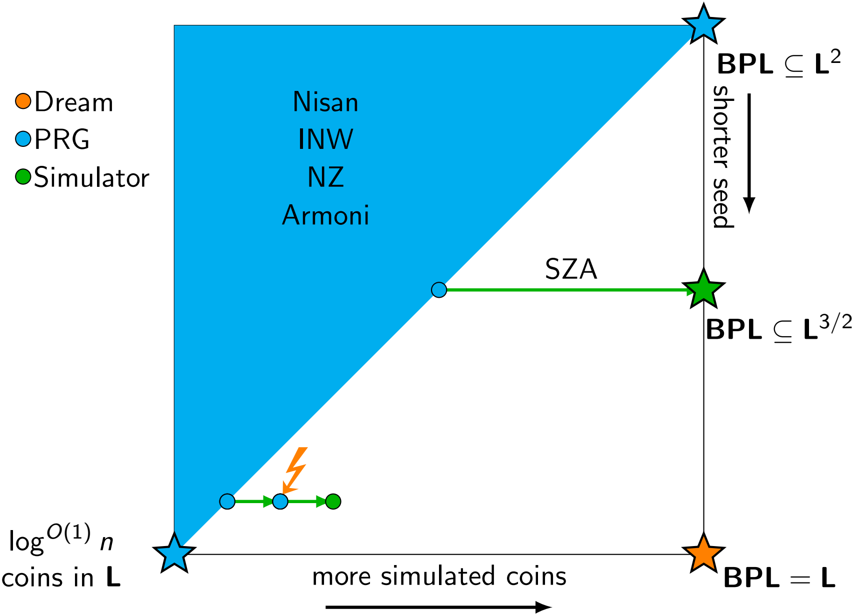
Proof of main result



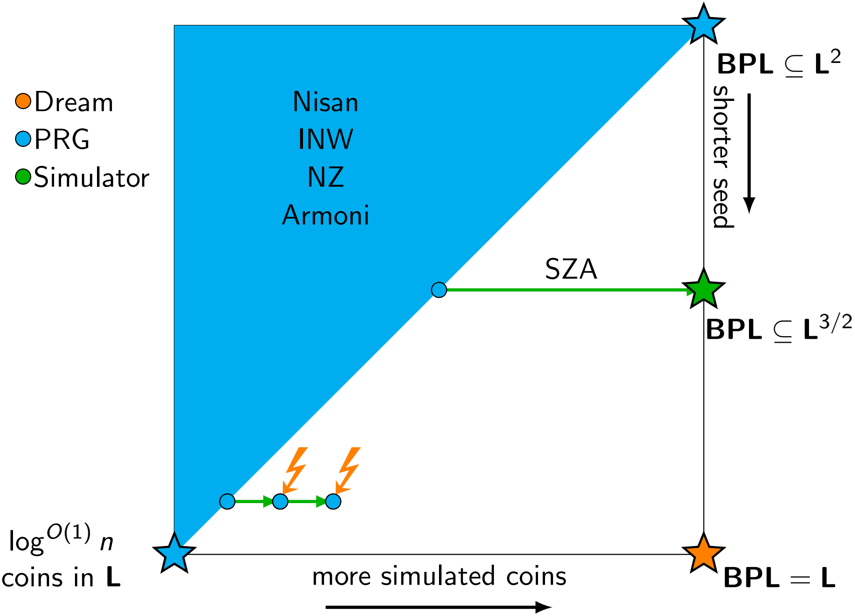
Proof of main result



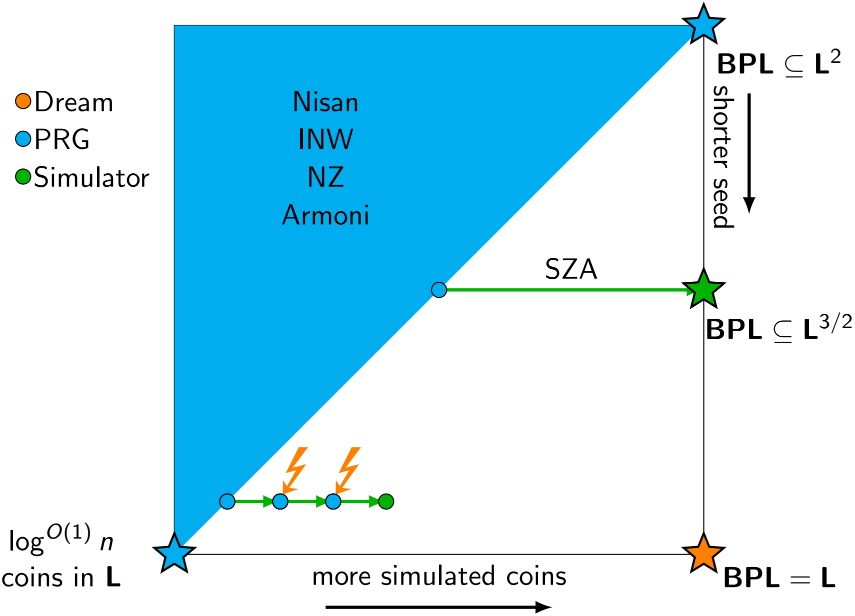
Proof of main result



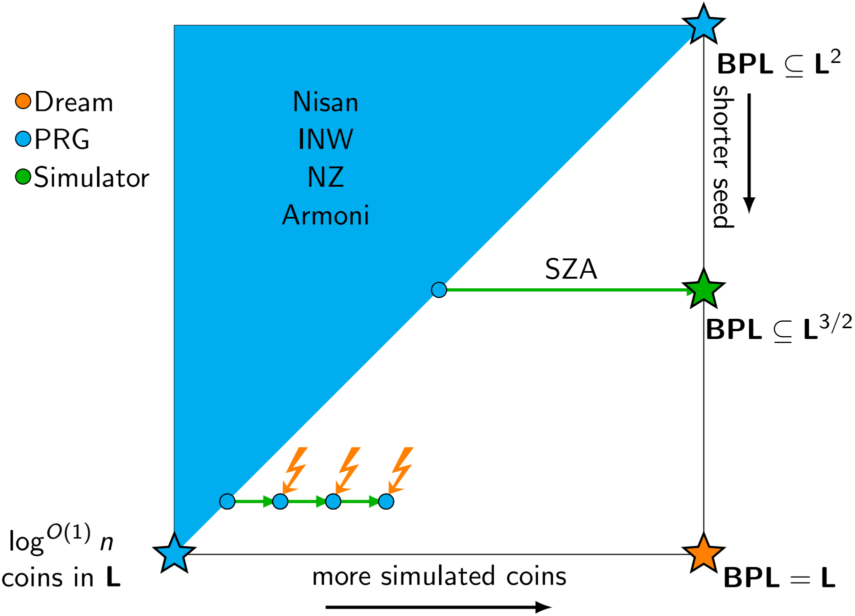
Proof of main result



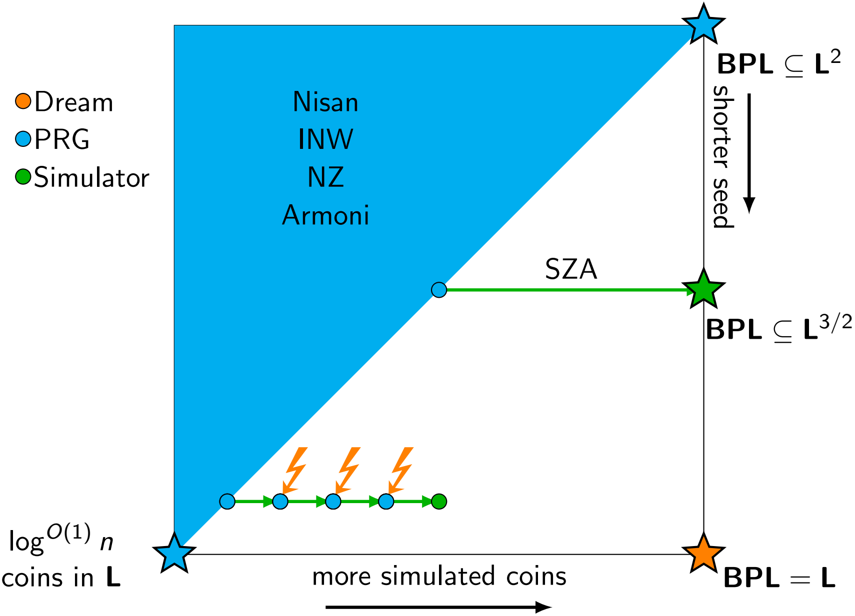
Proof of main result



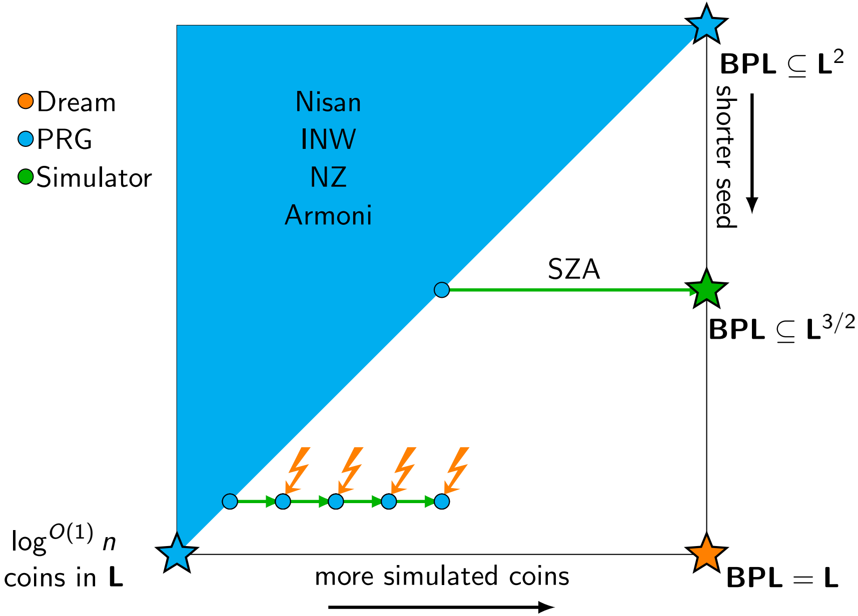
Proof of main result



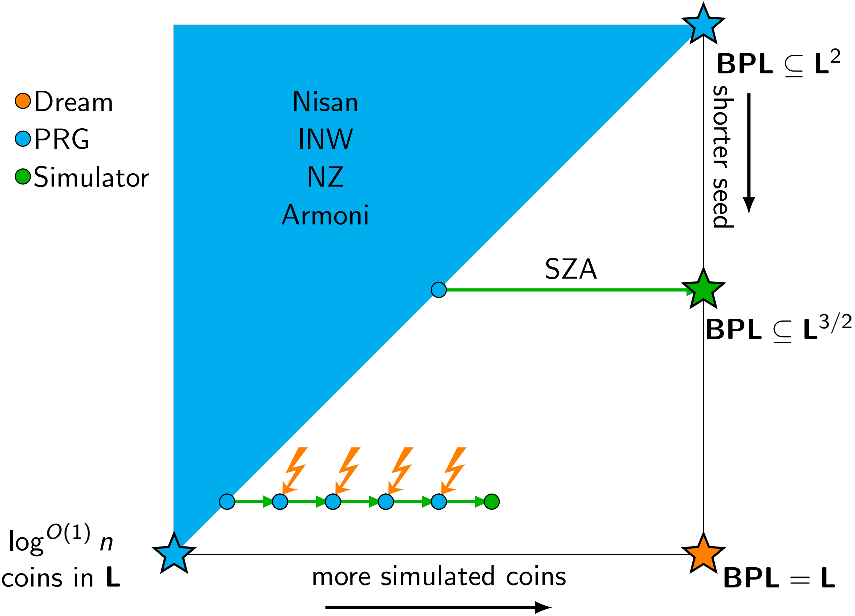
Proof of main result



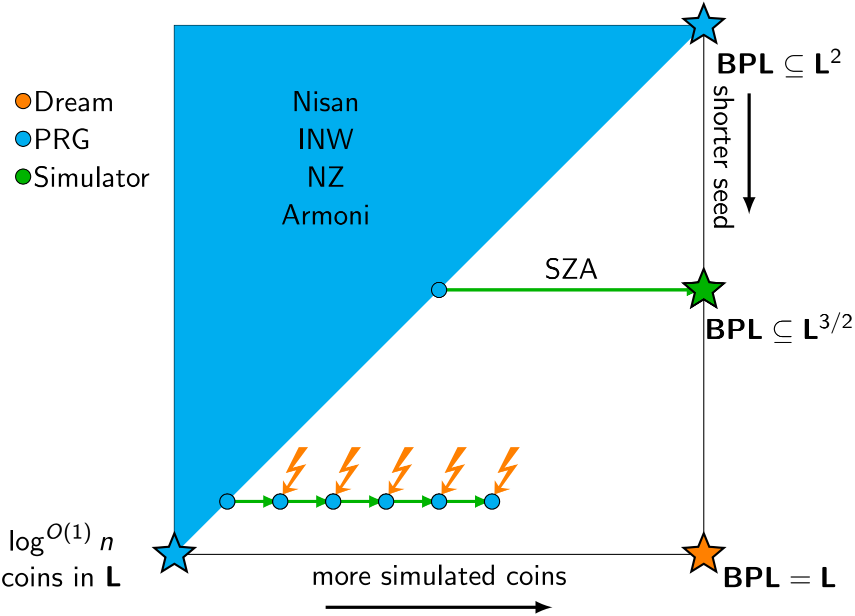
Proof of main result



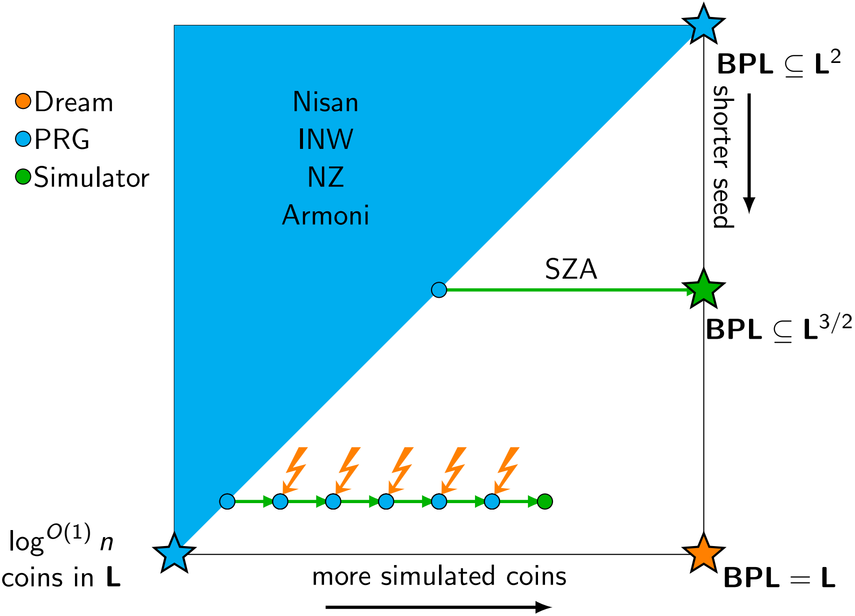
Proof of main result



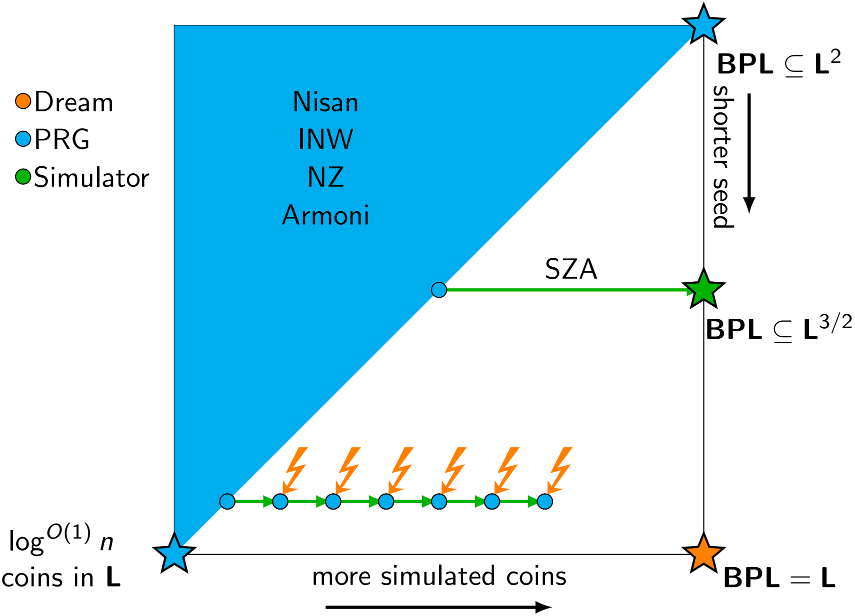
Proof of main result



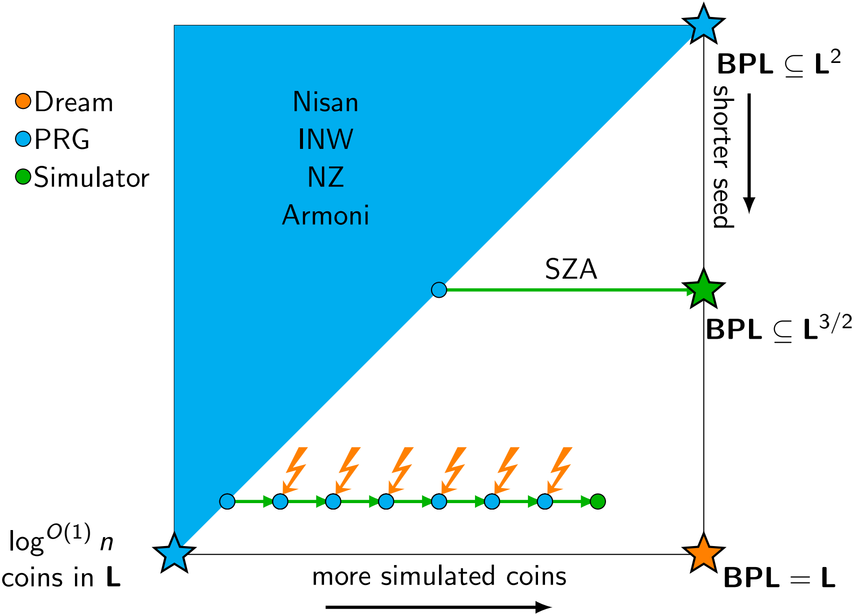
Proof of main result



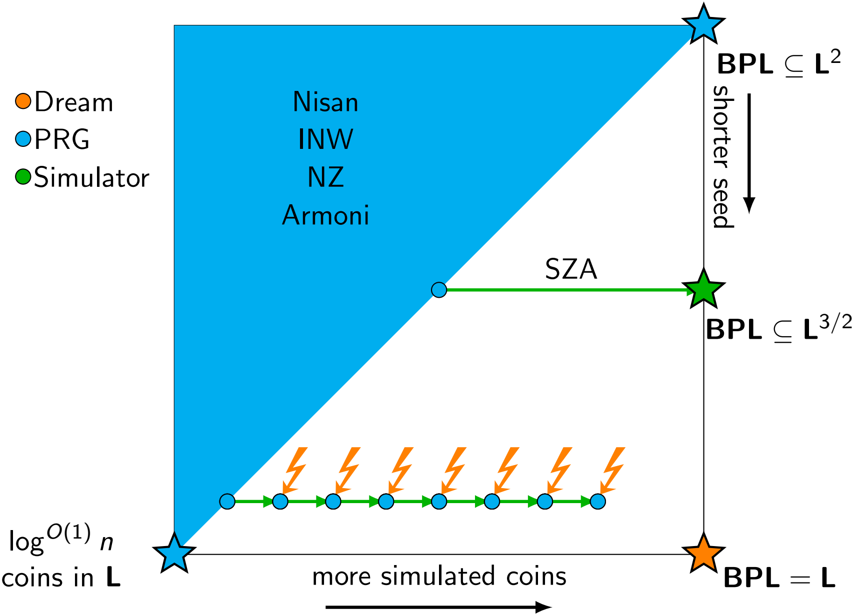
Proof of main result



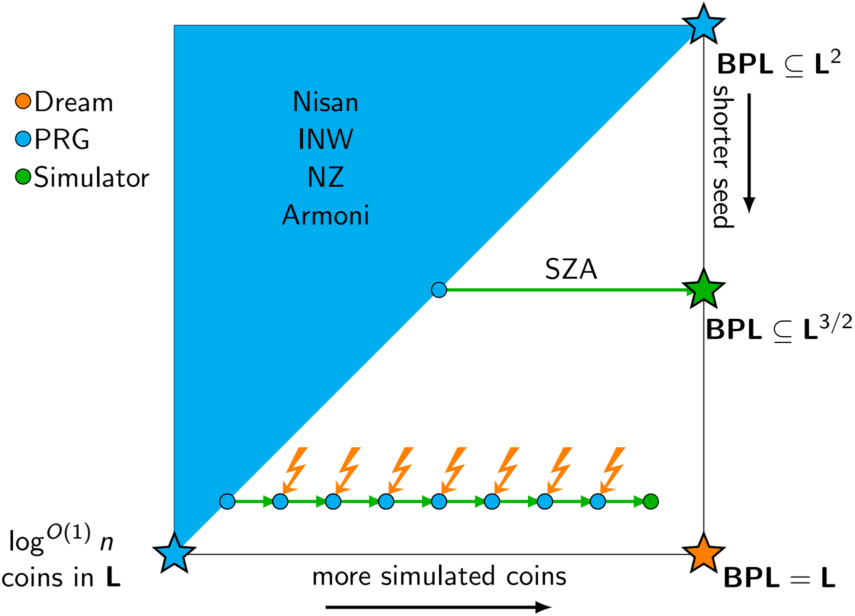
Proof of main result



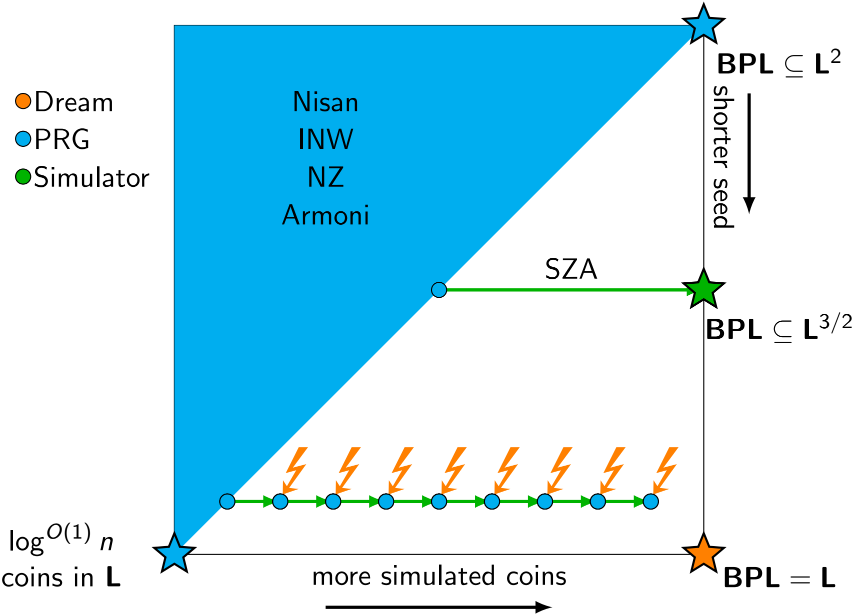
Proof of main result



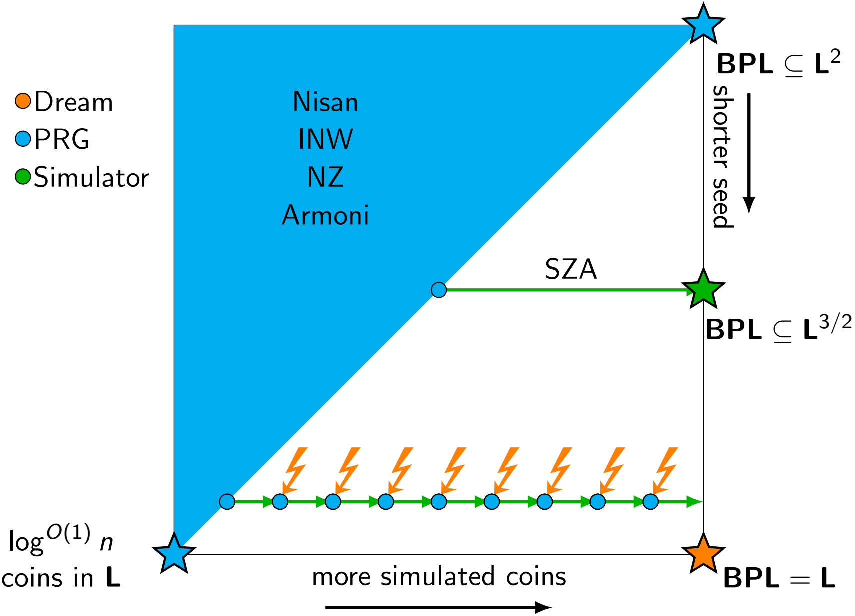
Proof of main result



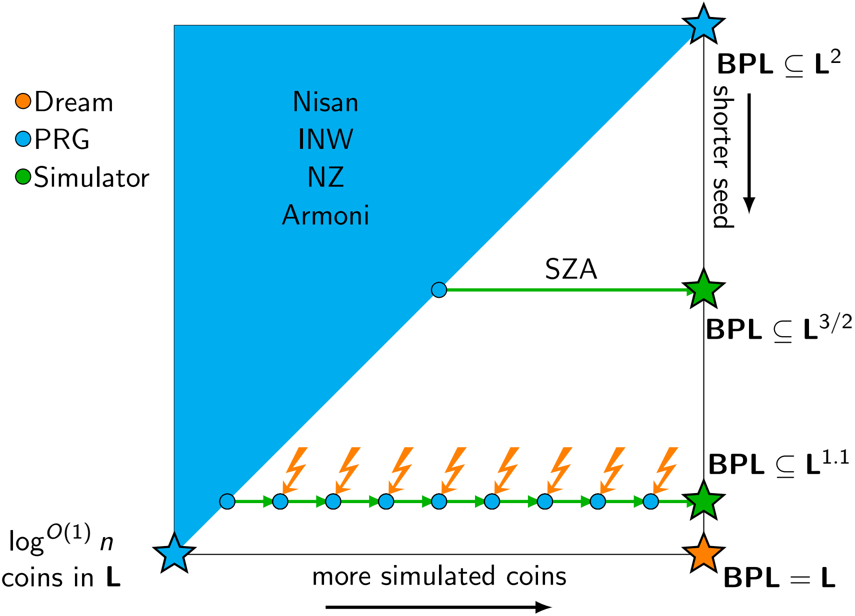
Proof of main result



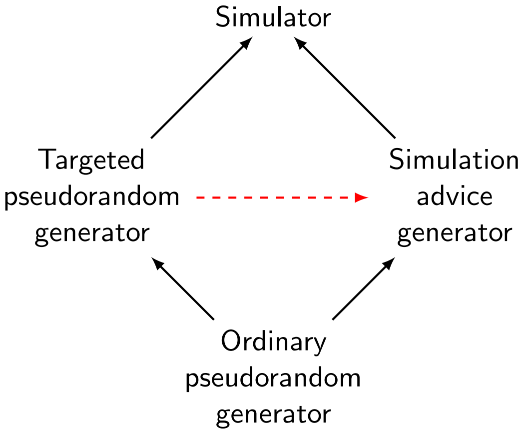
Proof of main result



Proof of main result



Stronger version of main result



Theorem: Dashed arrow transformation exists if and only if

$$\bigcap_{\alpha>0} \textbf{promise-BPSPACE}(\log^{1+\alpha} n) = \bigcap_{\alpha>0} \textbf{promise-DSPACE}(\log^{1+\alpha} n)$$

Conclusion

- This material is based upon work supported by
  - NSF GRFP Grant No. DGE-1610403
  - NSF Grant No. NSF CCF-1423544
- Thanks for your attention!
- Any questions?