High-performance pseudorandomness

Rasmus Pagh

IT UNIVERSITY OF COPENHAGEN

Work supported by:
Agenda

• Basic motivation and definitions.

• Great theoretical results - useless in practice?

• **New developments:**
  - Simple tabulation hashing.
  - Semi-explicit unbalanced expanders.
  - Pseudorandom generation.

• Open questions.
Motivating application:

Load balancing

• **Goal:**
  Distribute an unknown, possibly dynamic, set of items approximately evenly to a set of buckets.
Motivating application: 
Load balancing

• **Goal:** Distribute an unknown, possibly dynamic, set of items approximately evenly to a set of buckets.

• **Examples:** Hash tables, SSDs, distributed storage, distributed computation, network routing, parallel algorithms, …
Motivating application: Load balancing

- **Goal:** Distribute an unknown, possibly dynamic, set of items approximately evenly to a set of buckets.

- **Examples:** Hash tables, SSDs, distributed storage, distributed computation, network routing, parallel algorithms, ...

- **Main tool:** Random choice of assignment.
Motivating application: Load balancing

- **Goal:** Distribute an unknown, possibly dynamic, set of items approximately evenly to a set of buckets.

- **Examples:** Hash tables, SSDs, distributed storage, distributed computation, network routing, parallel algorithms, …

- **Main tool:** Random choice of assignment.
Motivating application: Load balancing

- **Goal:** Distribute an unknown, possibly dynamic, set of items approximately evenly to a set of buckets.
- **Examples:** Hash tables, SSDs, distributed storage, distributed computation, network routing, parallel algorithms, ...
- **Main tool:** Random choice of assignment.

This talk: Items are strings in $\Sigma^n$
Motivating application: Load balancing

• **Goal:**
  Distribute an unknown, possibly dynamic, set of items approximately evenly to a set of buckets.

• **Examples:** Hash tables, SSDs, distributed storage, distributed computation, network routing, parallel algorithms, …

• **Main tool:** Random choice of assignment.

This talk: Items are strings in $\Sigma^n$.

No significant loss of generality: Can map to (say) 16-byte strings with extremely small risk of collision.
Two interfaces to pseudorandomness

• **Basic object:**
  
  Function $f_s: \Sigma^n \rightarrow D$, indexed by a random seed $s$.  
  (Often implemented with “salt”: $f_s(x) = \text{hash}(s;x)$.)

1. **Random access** (hashing):
   
   Compute $f_s(x)$ for a given string $x$.

2. **Sequential access** (pseudorandom generator):
   
   Generate some sequence $f_s(x_1), f_s(x_2), \ldots$
What does theory say?
What does theory say?

On Randomness in Hash Functions

Martin Dietzfelbinger
Technische Universität Ilmenau
Linear hash functions

- NodeIndex hashfunc(int a, short c)
  
  ```
  { return ((1+8+64)*a + (1+4+16)*c) % HASHSIZE; }
  ```

  HASHSIZE is prime
Linear hash functions

- Associate key \( x \) with a vector \( v_x \) in \( \mathbb{F}_k \), field \( \mathbb{F} \).

For random \( a \) in \( \mathbb{F}_k \), compute \( a \cdot x \).

```c
NodeIndex hashfunc(int a, short c)
{
    return ((1+8+64)*a + (1+4+16)*c) % HASHSIZE;
}

HASHSIZE is prime
```
Linear hash functions

- Associate key $x$ with a vector $v_x$ in $\mathbb{F}^k$, field $\mathbb{F}$.
  For random $a$ in $\mathbb{F}^k$, compute $a \cdot x$.

This talk: $\mathbb{F}$ is a field with constant time operations
Linear hash functions

• Associate key $x$ with a vector $v_x$ in $\mathbb{F}^k$, field $\mathbb{F}$.
  For random $a$ in $\mathbb{F}^k$, compute $a \cdot x$.

• More examples:

This talk: $\mathbb{F}$ is a field with constant time operations
Linear hash functions

- Associate key $x$ with a vector $v_x$ in $\mathbb{F}^k$, field $\mathbb{F}$.
  
  For random $a$ in $\mathbb{F}^k$, compute $a \cdot x$.

- More examples:
  
  - $x$ in $\mathbb{F}$, $v_x = (x^0, x^1, x^2, \ldots, x^{k-1})$.

 Polynomial hash functions.

This talk: $\mathbb{F}$ is a field with constant time operations.
Linear hash functions

• Associate key \( x \) with a vector \( v_x \) in \( \mathbb{F}^k \), field \( \mathbb{F} \).
  
  For random \( a \) in \( \mathbb{F}^k \), compute \( a \cdot x \).

• More examples:
  - \( x \) in \( \mathbb{F} \), \( v_x = (x^0, x^1, x^2, \ldots, x^{k-1}) \).
    
    Polynomial hash functions.
  
  - \( x=x_1 \ldots x_n \), \( v_x = \) indicator vector of \( \{(i,x_i), i=1,\ldots,n\} \).
    
    Tabulation hashing.

This talk: \( \mathbb{F} \) is a field with constant time operations.
Linear hash functions

- Associate key $x$ with a vector $v_x$ in $F^k$, field $F$. For random $a$ in $F^k$, compute $a \cdot x$.

- More examples:
  - $x$ in $F$, $v_x = (x^0, x^1, x^2, \ldots, x^{k-1})$.
    Polynomial hash functions.
  - $x=x_1 \ldots x_n$, $v_x =$ indicator vector of $\{(i,x_i), i=1,\ldots,n\}$.
    Tabulation hashing.
  - $v_x =$ indicator vector of $\Gamma(x)$, $\Gamma$ neighbor function of a unique neighbor expander. “Siegel hashing.”

This talk: $F$ is a field with constant time operations.
Linear hash functions

• Associate key $x$ with a vector $v_x$ in $\mathbb{F}^k$, field $\mathbb{F}$.
  For random $a$ in $\mathbb{F}^k$, compute $a \cdot x$.

• More examples:
  - $x$ in $\mathbb{F}$, $v_x = (x^0, x^1, x^2, \ldots, x^{k-1})$.

Property: If every set of $k$ vectors $v_x$ is linearly independent, the hash function is $k$-independent

- $v_x = \text{indicator vector of } \Gamma(x), \Gamma \text{ neighbor function of a unique neighbor expander. } \text{“Siegel hashing.”} \)
Two famous, theoretically good linear hash functions

• **Universal hashing** [Carter & Wegman ’77]
  Small *expected* load for bucket of every item $i$.
  Aka. pairwise independence. Provably works for fingerprinting, hash tables with chaining, Bloom filters,…

• **$\epsilon$-independence** [Siegel et al. ’89, ‘93]:
  Bucket load obeys Chernoff concentration bounds.
  Provably works for load balancing, cuckoo hashing, linear probing, double hashing,…

• Both have constant evaluation time, but seem to be little used in practice. **Why?**
On Randomness in Hash Functions

Martin Dietzfelbinger
Technische Universität Ilmenau
On Randomness in Hash Functions

Martin Dietzfelbinger
Technische Universität Ilmenau

RAM model of computation
Issues typically neglected in theory of hashing

Things that go beyond the word RAM model:

• Memory hierarchy effects
• Contemporary instruction sets
• Pipelining and small-scale parallelism
Other problems in practice

• Universal hashing:
  - Requires seed length proportional to $\log n$.
  - Computationally efficient implementations tend to use $O(n \log n)$ seed length.

Items are strings in $\Sigma^n$
Other problems in practice

- **Universal hashing:**
  - Requires seed length proportional to $\log n$.
  - Computationally efficient implementations tend to use $O(n \log n)$ seed length.

- **$u^\varepsilon$-independence:**
  - Siegel’s method requires a very large (constant) evaluation time and relatively large space.
  - (More efficient solutions would be possible given access to efficient *unbalanced expanders.*)
Prevalent in practice: Deterministic hashing
Prevalent in practice: Deterministic hashing

- Java string hashing (signed 32-bit arithmetic):
  \[ h(a_1a_2\ldots a_n) = a_n + 31 \cdot h(a_1a_2\ldots a_{n-1}) \]
Prevalent in practice: Deterministic hashing

- Java string hashing (signed 32-bit arithmetic):
  \[ h(a_1a_2\ldots a_n) = a_n + 31 \; h(a_1a_2\ldots a_{n-1}) \]

- Collisions:
  - \( h(Aa) = h(BB) = 2112 \) (equivalent substrings)
  - \( h(AaAa) = h(AaBB) = h(BBAa) = h(BBBB) = 2095104 \)
  - ...

Prevalent in practice: Deterministic hashing

- Java string hashing (signed 32-bit arithmetic):
  \[ h(a_1a_2\ldots a_n) = a_n + 31 \cdot h(a_1a_2\ldots a_{n-1}) \]

- Collisions:
  - \( h(Aa) = h(BB) = 2112 \) (equivalent substrings)
  - \( h(AaAa) = h(AaBB) = h(BBAa) = h(BBBB) = 2095104 \)
  - ...

- Recent heuristic hash functions, with focus on evaluation time: MurmurHash, CityHash, SipHash.
But (some) people are starting to care!
But (some) people are starting to care!

• Crosby & Wallach: Denial of Service via Algorithmic Complexity Attacks. Usenix Security ’03.
  - Follow-ups: Chaos Communication Congress ’11, ’12.
  - Meet-in-the-middle (birthday) attacks find collisions among $b$-bit hash values in time and space $O(2^{b/2})$. 
But (some) people are starting to care!

• Crosby & Wallach: *Denial of Service via Algorithmic Complexity Attacks*. Usenix Security ’03.
  - Follow-ups: Chaos Communication Congress ’11, ’12.
  - Meet-in-the-middle (birthday) attacks find collisions among $b$-bit hash values in time and space $O(2^{b/2})$.

• Java, C++, C# libraries still use deterministic hashing.
  - But Java falls back to BST for long hash chains!
But (some) people are starting to care!

- Crosby & Wallach: *Denial of Service via Algorithmic Complexity Attacks*. Usenix Security ’03.
  - Follow-ups: Chaos Communication Congress ’11, ’12.
  - Meet-in-the-middle (birthday) attacks find collisions among \( b \)-bit hash values in time and space \( O(2^{b/2}) \).

- Java, C++, C# libraries still use deterministic hashing.
  - But Java falls back to BST for long hash chains!

- **NEW**: Ruby 1.9, Python 3.3, [Perl 5.18] now use *random hashing* [if deterministic hashing fails].
Agenda

• Basic definitions and motivation.

• Great theoretical results - useless in practice?

• New developments:
  - Simple tabulation hashing.
  - Semi-explicit unbalanced expanders.
  - Pseudorandom generation.

• Open questions.
Simple tabulation hashing

• Let \( h_1, \ldots, h_n : \Sigma \rightarrow D \) be fully random functions (implemented as lookup tables).

• Simple tabulation [Zobrist '70]:
  \[
  h(x_1 \ldots x_n) = h_1(x_1) \oplus \ldots \oplus h_n(x_n).
  \]
Simple tabulation hashing

• Let $h_1,\ldots,h_n : \Sigma \rightarrow D$ be fully random functions (implemented as lookup tables).

• Simple tabulation [Zobrist ’70]:

  - $h(x_1\ldots x_n) = h_1(x_1) \oplus \ldots \oplus h_n(x_n)$.  \(\oplus\) is bitwise XOR
Simple tabulation hashing

• Let $h_1,\ldots,h_n: \Sigma \rightarrow D$ be fully random functions (implemented as lookup tables).

• Simple tabulation [Zobrist ’70]:
  
  - $h(x_1\ldots x_n) = h_1(x_1) \oplus \ldots \oplus h_n(x_n)$. $\oplus$ is bitwise XOR
  
  - Only 3-independent.
Simple tabulation hashing

• Let $h_1, \ldots, h_n: \Sigma \rightarrow D$ be fully random functions (implemented as lookup tables).

• Simple tabulation [Zobrist ’70]:
  - $h(x_1 \ldots x_n) = h_1(x_1) \oplus \ldots \oplus h_n(x_n)$. \(\oplus\) is bitwise XOR
  - Only 3-independent.
  - [Pătrașcu-Thorup, J. ACM ’12]: Yields Chernoff bounds for constant $n$ (space $n\Sigma$).
The power of simple tabulation hashing

- Gives many properties of full randomness
  - But not all, e.g., dynamic cuckoo hashing is slow for some data.
Double tabulation

Mikkel Thorup on TV
Double tabulation

SWAT 2014
14th Scandinavian Symposium and Workshops
July 2-4 2014, Copenhagen, Denmark

Thursday, July 3rd 2014
Breakfast
Invited Speaker: Mikkel Thorup
Double tabulation

- Let $h_1, \ldots, h_n: \Sigma \rightarrow \Sigma^6$, $g_1, \ldots, g_{6n}: \Sigma \rightarrow D$ be fully random functions (impl. as lookup tables).

- Double tabulation:
  - $(y_1 \ldots y_{6n}) = h_1(x_1) \oplus \ldots \oplus h_n(x_n)$.
  - $h(x_1 \ldots x_n) = g_1(y_1) \oplus \ldots \oplus g_{6n}(y_{6n})$.

[Thorup, FOCS ’13]
Double tabulation

- Let $h_1, \ldots, h_n: \Sigma \rightarrow \Sigma^6$, $g_1, \ldots, g_{6n}: \Sigma \rightarrow D$ be fully random functions (impl. as lookup tables).

- Double tabulation:
  - $(y_1 \ldots y_{6n}) = h_1(x_1) \oplus \ldots \oplus h_n(x_n)$.
  - $h(x_1 \ldots x_n) = g_1(y_1) \oplus \ldots \oplus g_{6n}(y_{6n})$.

- **Main result**: Gives $k = \Sigma^{\Omega(1)}$ independence! (whp. over choice of $h_1, \ldots, h_n$)

[Thorup, FOCS '13]
Why double tabulation gives high independence

\[ \sum^n \quad \sum^{6n} \quad D \]

- \( \sum^n \) tabulation hashing
- \( \sum^{6n} \) tabulation hashing
- \( D \)
Why double tabulation gives high independence

\[ \Sigma^n \quad \Sigma \times \Sigma \times \ldots \times \Sigma \quad D \]

tabulation

hashing
Why double tabulation gives high independence

\[ \Sigma^n \rightarrow \Sigma \times \Sigma \times \ldots \times \Sigma \rightarrow D \]

[Thorup '13]: Encodes a unique-neighbor expander with high probability
Why double tabulation gives high independence

[Thorup ’13]:
Encodes a unique-neighbor expander with high probability
Why double tabulation gives high independence

\[ \sum^n \equiv \sum \times \sum \times \ldots \times \sum \equiv D \]

[Thorup '13]:
Encodes a unique-neighbor expander with high probability
Why double tabulation gives high independence

\[ \sum^n x \times \sum \times \ldots \times \sum \longleftrightarrow D \]

[Thurup '13]:
Encodes a unique-neighbor expander with high probability
Why double tabulation gives high independence

[Thorup ’13]: Encodes a unique-neighbor expander with high probability

\[ \Sigma^n \rightarrow \Sigma \times \Sigma \times \ldots \times \Sigma \rightarrow D \]

unique
Why double tabulation gives high independence

[Thorup '13]: Encodes a unique-neighbor expander with high probability

\[ \Sigma^n \times \Sigma \times \ldots \times \Sigma \]

\[ D \]

unique

independent of other values

[Thorup '13]: Encodes a unique-neighbor expander with high probability
Why double tabulation gives high independence

[Thorup '13]: Encodes a unique-neighbor expander with high probability

\[ \Sigma^n \quad \Sigma \times \Sigma \times ... \times \Sigma \quad D \]

unique

\[ \sum x \]

\[ \rightarrow \text{every set of } k \text{ vectors } v_x \text{ lin. indep.} \]

\[ \rightarrow \text{hash function is } k\text{-independent} \]

independent of other values
Why double tabulation gives high independence

[Thorup '13]: Encodes a unique-neighbor expander with high probability

\[ \sum^n \times \sum \times \ldots \times \sum \]

\[ D \]

"Semi-explicit" expander: Description needs many random bits, but much fewer than an adjacency list representation.

⇒ every set of \( k \) vectors \( v_x \) lin. indep.
⇒ hash function is \( k \)-independent
How fast is it?

- Time-space trade-off.

- Parameters in [Thorup ’13], 100-independence:
  - 32-bit keys: 22 fast lookups (in L3 cache).
  - 64-bit keys: 27 slow lookups (out of cache) or 52 fast lookups (in L3 cache).
How fast is it?

- Time-space trade-off.
- Parameters in [Thorup ’13], 100-independence:
  - 32-bit keys: 22 fast lookups (in L3 cache).
  - 64-bit keys: 27 slow lookups (out of cache) or 52 fast lookups (in L3 cache).
- Typical latency:
  - L3 cache access 3 ns.
  - RAM memory access 15 ns.
How fast is it?

- Time-space trade-off.

- Parameters in [Thorup ’13], 100-independence:
  - 32-bit keys: 22 fast lookups ~ 60 ns
  - 64-bit keys: 27 slow lookups ~ 400 ns or 52 fast lookups ~ 150 ns

- Typical latency:
  - L3 cache access 3 ns.
  - RAM memory access 15 ns.
Comparing to polynomial hashing
Comparing to polynomial hashing

- Polynomial $k$-independent hash function (assuming key $x$ from field $F$):

$$p(x) = \sum_{i=0}^{k-1} a_i x^i$$
Comparing to polynomial hashing

- Polynomial $k$-independent hash function (assuming key $x$ from field $F$):

$$p(x) = \sum_{i=0}^{k-1} a_i x^i$$

- Divide-and-conquer Horner’s rule:

$$p(x) = x p_{\text{odd}}(x^2) + p_{\text{even}}(x^2)$$

Reduces data dependencies!
Implementing field operations

- For GF($2^{64}$): Use new CMUL instruction with sparse irreducible polynomial.

work by Tobias Christiani
Implementing field operations

- For $\text{GF}(2^{64})$: Use new CMUL instruction with sparse irreducible polynomial.
  - Time for 100-independence ca. 300 ns
Implementing field operations

- For GF($2^{64}$): Use new CMUL instruction with sparse irreducible polynomial.
  - Time for 100-independence ca. 300 ns

- For GF($2^{61}-1$): Use 128-bit registers and special code for modulo (Mersenne prime).

work by Tobias Christiani
Implementing field operations

- For GF($2^{64}$): Use new CMUL instruction with sparse irreducible polynomial.
  - Time for 100-independence ca. 300 ns

- For GF($2^{61}$-1): Use 128-bit registers and special code for modulo (Mersenne prime).
  - Time for 100-independence ca. 100 ns
Implementing field operations

- For $GF(2^{64})$: Use new CMUL instruction with sparse irreducible polynomial.
  - Time for 100-independence ca. 300 ns
- For $GF(2^{61}-1)$: Use 128-bit registers and special code for modulo (Mersenne prime).
  - Time for 100-independence ca. 100 ns

Double tabulation hashing wins for 32-bit keys, loses for 64-bit keys!
Agenda

• Basic definitions and motivation.

• Great theoretical results - useless in practice?

• New developments:
  - Simple tabulation hashing.
  - Semi-explicit unbalanced expanders.
  - Pseudorandom generation.

• Open questions.
Explicit constructions of $O(1)$-degree expanders

- Known to exist: Degree 3, $V_{\text{right}} < (V_{\text{left}})^{3/4}$, $k=(V_{\text{left}})^{1/4}$
- Explicitly [Capalbo '02]: Degree 3, $V_{\text{right}} < (21/22) V_{\text{left}}$
Explicit constructions of O(1)-degree expanders

- Known to exist: Degree 3, $V_{\text{right}} < (V_{\text{left}})^{3/4}$, $k=(V_{\text{left}})^{1/4}$
- Explicitly [Capalbo ’02]: Degree 3, $V_{\text{right}} < (21/22) V_{\text{left}}$

Can turn $n$ numbers that are $k$-independent into $(22/21)n$ numbers that are $k/3$-independent.
Can turn $n$ numbers that are $k$-independent into $(22/21)n$ numbers that are $k/3$-independent

By repetition, can turn $k$ numbers that are $k$-independent into $k\text{ polylog } k$ numbers that are $k/\text{polylog } k$-independent
Combining with FFT

• Using FFT it is possible to evaluate a degree-$d$ polynomial on $d$ inputs in $\sim d \log^2 d$ time.
New generator

- **Theorem**: Given preprocessing time and space $k \text{ polylog } k$ we can generate a sequence of $k$-independent random variables from $F$ of length up to $|F|$ in $O(1)$ time/value.

  joint work with Tobias Christiani, to appear at FOCS 2014
Experimental results
Assume that expander graph is cheaply available (e.g. read edges from SSD)
Assume that expander graph is cheaply available (e.g. read edges from SSD).

### Experimental results

<table>
<thead>
<tr>
<th>$k$</th>
<th>Horner ns</th>
<th>FFT ns</th>
<th>$c$</th>
<th>$m$</th>
<th>$d$</th>
<th>$\delta$</th>
<th>time/value ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^5$</td>
<td>177</td>
<td>243</td>
<td>64</td>
<td>$2^{13}$</td>
<td>8</td>
<td>$10^{-7}$</td>
<td>15</td>
</tr>
<tr>
<td>$2^6$</td>
<td>361</td>
<td>294</td>
<td>64</td>
<td>$2^{14}$</td>
<td>8</td>
<td>$10^{-8}$</td>
<td>16</td>
</tr>
<tr>
<td>$2^7$</td>
<td>730</td>
<td>338</td>
<td>64</td>
<td>$2^{15}$</td>
<td>8</td>
<td>$10^{-9}$</td>
<td>19</td>
</tr>
<tr>
<td>$2^8$</td>
<td>1470</td>
<td>375</td>
<td>64</td>
<td>$2^{16}$</td>
<td>8</td>
<td>$10^{-10}$</td>
<td>23</td>
</tr>
<tr>
<td>$2^9$</td>
<td>2950</td>
<td>412</td>
<td>64</td>
<td>$2^{17}$</td>
<td>8</td>
<td>$10^{-11}$</td>
<td>24</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>5902</td>
<td>449</td>
<td>64</td>
<td>$2^{18}$</td>
<td>8</td>
<td>$10^{-12}$</td>
<td>25</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>11808</td>
<td>487</td>
<td>64</td>
<td>$2^{18}$</td>
<td>8</td>
<td>$10^{-12}$</td>
<td>35</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>23627</td>
<td>523</td>
<td>32</td>
<td>$2^{18}$</td>
<td>8</td>
<td>$10^{-12}$</td>
<td>43</td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>47183</td>
<td>561</td>
<td>32</td>
<td>$2^{18}$</td>
<td>16</td>
<td>$10^{-29}$</td>
<td>54</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>94429</td>
<td>599</td>
<td>64</td>
<td>$2^{22}$</td>
<td>8</td>
<td>$10^{-15}$</td>
<td>68</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>188258</td>
<td>638</td>
<td>64</td>
<td>$2^{23}$</td>
<td>8</td>
<td>$10^{-16}$</td>
<td>69</td>
</tr>
</tbody>
</table>
### Experimental results

<table>
<thead>
<tr>
<th>$k$</th>
<th>Horner ns</th>
<th>FFT ns</th>
<th>$c$</th>
<th>$m$</th>
<th>$d$</th>
<th>$\delta$</th>
<th>ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^5$</td>
<td>177</td>
<td>243</td>
<td>64</td>
<td>$2^{13}$</td>
<td>8</td>
<td>$10^{-7}$</td>
<td>15</td>
</tr>
<tr>
<td>$2^6$</td>
<td>361</td>
<td>294</td>
<td>64</td>
<td>$2^{14}$</td>
<td>8</td>
<td>$10^{-8}$</td>
<td>16</td>
</tr>
<tr>
<td>$2^7$</td>
<td>730</td>
<td>338</td>
<td>64</td>
<td>$2^{15}$</td>
<td>8</td>
<td>$10^{-9}$</td>
<td>19</td>
</tr>
<tr>
<td>$2^8$</td>
<td>1470</td>
<td>375</td>
<td>64</td>
<td>$2^{16}$</td>
<td>8</td>
<td>$10^{-10}$</td>
<td>23</td>
</tr>
<tr>
<td>$2^9$</td>
<td>2950</td>
<td>412</td>
<td>64</td>
<td>$2^{17}$</td>
<td>8</td>
<td>$10^{-11}$</td>
<td>24</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>5902</td>
<td>449</td>
<td>64</td>
<td>$2^{18}$</td>
<td>8</td>
<td>$10^{-12}$</td>
<td>25</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>11808</td>
<td>487</td>
<td>32</td>
<td>$2^{18}$</td>
<td>8</td>
<td>$10^{-12}$</td>
<td>35</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>23627</td>
<td>523</td>
<td>64</td>
<td>$2^{18}$</td>
<td>16</td>
<td>$10^{-29}$</td>
<td>43</td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>47183</td>
<td>561</td>
<td>32</td>
<td>$2^{18}$</td>
<td>16</td>
<td>$10^{-29}$</td>
<td>54</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>94429</td>
<td>599</td>
<td>64</td>
<td>$2^{22}$</td>
<td>8</td>
<td>$10^{-15}$</td>
<td>68</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>188258</td>
<td>638</td>
<td>64</td>
<td>$2^{23}$</td>
<td>8</td>
<td>$10^{-16}$</td>
<td>69</td>
</tr>
</tbody>
</table>
### Experimental results

<table>
<thead>
<tr>
<th>$k$</th>
<th>Horner ns</th>
<th>FFT ns</th>
<th>$c$</th>
<th>$m$</th>
<th>$d$</th>
<th>$\delta$</th>
<th>ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^5$</td>
<td>177</td>
<td>243</td>
<td>64</td>
<td>2$^{13}$</td>
<td>8</td>
<td>$10^{-7}$</td>
<td>15</td>
</tr>
<tr>
<td>$2^6$</td>
<td>361</td>
<td>294</td>
<td>64</td>
<td>2$^{14}$</td>
<td>8</td>
<td>$10^{-8}$</td>
<td>16</td>
</tr>
<tr>
<td>$2^7$</td>
<td>730</td>
<td>338</td>
<td>64</td>
<td>2$^{15}$</td>
<td>8</td>
<td>$10^{-9}$</td>
<td>19</td>
</tr>
<tr>
<td>$2^8$</td>
<td>1470</td>
<td>375</td>
<td>64</td>
<td>2$^{16}$</td>
<td>8</td>
<td>$10^{-10}$</td>
<td>23</td>
</tr>
<tr>
<td>$2^9$</td>
<td>2950</td>
<td>412</td>
<td>64</td>
<td>2$^{17}$</td>
<td>8</td>
<td>$10^{-11}$</td>
<td>24</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>5902</td>
<td>449</td>
<td>64</td>
<td>2$^{18}$</td>
<td>8</td>
<td>$10^{-12}$</td>
<td>25</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>11808</td>
<td>487</td>
<td>32</td>
<td>2$^{18}$</td>
<td>8</td>
<td>$10^{-12}$</td>
<td>35</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>23627</td>
<td>523</td>
<td>64</td>
<td>2$^{18}$</td>
<td>16</td>
<td>$10^{-29}$</td>
<td>43</td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>47183</td>
<td>561</td>
<td>32</td>
<td>2$^{18}$</td>
<td>16</td>
<td>$10^{-29}$</td>
<td>54</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>94429</td>
<td>599</td>
<td>64</td>
<td>2$^{22}$</td>
<td>8</td>
<td>$10^{-15}$</td>
<td>68</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>188258</td>
<td>638</td>
<td>64</td>
<td>2$^{23}$</td>
<td>8</td>
<td>$10^{-16}$</td>
<td>69</td>
</tr>
</tbody>
</table>
Assume that expander graph is cheaply available (e.g. read edges from SSD).

**Experimental results**

<table>
<thead>
<tr>
<th>$k$</th>
<th>Horner</th>
<th>FFT</th>
<th>FFT + Expander</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ns</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>$2^5$</td>
<td>177</td>
<td>243</td>
<td>64 $2^{13}$ 8 $10^{-7}$ 15</td>
</tr>
<tr>
<td>$2^6$</td>
<td>361</td>
<td>294</td>
<td>64 $2^{14}$ 8 $10^{-8}$ 16</td>
</tr>
<tr>
<td>$2^7$</td>
<td>730</td>
<td>338</td>
<td>64 $2^{15}$ 8 $10^{-9}$ 19</td>
</tr>
<tr>
<td>$2^8$</td>
<td>1470</td>
<td>375</td>
<td>64 $2^{16}$ 8 $10^{-10}$ 23</td>
</tr>
<tr>
<td>$2^9$</td>
<td>2950</td>
<td>412</td>
<td>64 $2^{17}$ 8 $10^{-11}$ 24</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>5902</td>
<td>449</td>
<td>64 $2^{18}$ 8 $10^{-12}$ 25</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>11808</td>
<td>487</td>
<td>32 $2^{18}$ 8 $10^{-12}$ 35</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>23627</td>
<td>523</td>
<td>64 $2^{18}$ 16 $10^{-29}$ 43</td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>47183</td>
<td>561</td>
<td>32 $2^{18}$ 16 $10^{-29}$ 54</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>94429</td>
<td>599</td>
<td>64 $2^{22}$ 8 $10^{-15}$ 68</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>188258</td>
<td>638</td>
<td>64 $2^{23}$ 8 $10^{-16}$ 69</td>
</tr>
</tbody>
</table>
### Experimental results

<table>
<thead>
<tr>
<th>$k$</th>
<th>Horner ns</th>
<th>FFT ns</th>
<th>$c$</th>
<th>$m$</th>
<th>$d$</th>
<th>$\delta$</th>
<th>$\delta$</th>
<th>ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^5$</td>
<td>177</td>
<td>243</td>
<td>64</td>
<td>$2^{13}$</td>
<td>8</td>
<td>$10^{-7}$</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>$2^6$</td>
<td>361</td>
<td>294</td>
<td>64</td>
<td>$2^{14}$</td>
<td>8</td>
<td>$10^{-8}$</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>$2^7$</td>
<td>730</td>
<td>338</td>
<td>64</td>
<td>$2^{15}$</td>
<td>8</td>
<td>$10^{-9}$</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>$2^8$</td>
<td>1470</td>
<td>375</td>
<td>64</td>
<td>$2^{16}$</td>
<td>8</td>
<td>$10^{-10}$</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>$2^9$</td>
<td>2950</td>
<td>412</td>
<td>64</td>
<td>$2^{17}$</td>
<td>8</td>
<td>$10^{-11}$</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>5902</td>
<td>449</td>
<td>64</td>
<td>$2^{18}$</td>
<td>8</td>
<td>$10^{-12}$</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>11808</td>
<td>487</td>
<td>32</td>
<td>$2^{18}$</td>
<td>8</td>
<td>$10^{-12}$</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>23627</td>
<td>523</td>
<td>64</td>
<td>$2^{18}$</td>
<td>16</td>
<td>$10^{-29}$</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>47183</td>
<td>561</td>
<td>32</td>
<td>$2^{18}$</td>
<td>16</td>
<td>$10^{-29}$</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>94429</td>
<td>599</td>
<td>64</td>
<td>$2^{22}$</td>
<td>8</td>
<td>$10^{-15}$</td>
<td>68</td>
<td></td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>188258</td>
<td>638</td>
<td>64</td>
<td>$2^{23}$</td>
<td>8</td>
<td>$10^{-16}$</td>
<td>69</td>
<td></td>
</tr>
</tbody>
</table>
Assume that expander graph is cheaply available (e.g. read edges from SSD).

<table>
<thead>
<tr>
<th>$2^5$</th>
<th>177</th>
<th>243</th>
<th>64</th>
<th>$2^{13}$</th>
<th>8</th>
<th>$10^{-7}$</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^6$</td>
<td>361</td>
<td>294</td>
<td>64</td>
<td>$2^{14}$</td>
<td>8</td>
<td>$10^{-8}$</td>
<td>16</td>
</tr>
<tr>
<td>$2^7$</td>
<td>730</td>
<td>338</td>
<td>64</td>
<td>$2^{15}$</td>
<td>8</td>
<td>$10^{-9}$</td>
<td>19</td>
</tr>
<tr>
<td>$2^8$</td>
<td>1470</td>
<td>375</td>
<td>64</td>
<td>$2^{16}$</td>
<td>8</td>
<td>$10^{-11}$</td>
<td>24</td>
</tr>
<tr>
<td>$2^9$</td>
<td>2950</td>
<td>412</td>
<td>64</td>
<td>$2^{17}$</td>
<td>8</td>
<td>$10^{-12}$</td>
<td>25</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>5902</td>
<td>449</td>
<td>64</td>
<td>$2^{18}$</td>
<td>8</td>
<td>$10^{-12}$</td>
<td>35</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>11808</td>
<td>487</td>
<td>64</td>
<td>$2^{18}$</td>
<td>16</td>
<td>$10^{-29}$</td>
<td>43</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>23627</td>
<td>523</td>
<td>64</td>
<td>$2^{18}$</td>
<td>16</td>
<td>$10^{-29}$</td>
<td>54</td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>47183</td>
<td>561</td>
<td>64</td>
<td>$2^{22}$</td>
<td>8</td>
<td>$10^{-15}$</td>
<td>68</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>94429</td>
<td>599</td>
<td>64</td>
<td>$2^{23}$</td>
<td>8</td>
<td>$10^{-16}$</td>
<td>69</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>188258</td>
<td>638</td>
<td>64</td>
<td>$2^{24}$</td>
<td>8</td>
<td>$10^{-17}$</td>
<td>77</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>376143</td>
<td>678</td>
<td>64</td>
<td>$2^{25}$</td>
<td>8</td>
<td>$10^{-18}$</td>
<td>85</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>751781</td>
<td>719</td>
<td>64</td>
<td>$2^{26}$</td>
<td>8</td>
<td>$10^{-19}$</td>
<td>93</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>1505016</td>
<td>765</td>
<td>64</td>
<td>$2^{26}$</td>
<td>16</td>
<td>$10^{-46}$</td>
<td>110</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>3015969</td>
<td>808</td>
<td>32</td>
<td>$2^{26}$</td>
<td>8</td>
<td>$10^{-19}$</td>
<td>93</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>6082313</td>
<td>864</td>
<td>64</td>
<td>$2^{26}$</td>
<td>16</td>
<td>$10^{-46}$</td>
<td>175</td>
</tr>
</tbody>
</table>
Some open questions

• Can “semi-explicit” expanders make a practical impact in other settings where expander graphs are used?

• Is there a single method for, say, 100-independence that is superior for all key lengths?
Thanks

The following people contributed information for preparing this talk:

• Tobias Christiani, Martin Dietzfelbinger, Daniel Lemire, Ilya Mironov, Udi Wieder