A battle of bits: building confidence in cryptography

D. J. Bernstein
University of Illinois at Chicago

Tanja Lange
Technische Universiteit Eindhoven

Negation joint work with:
Peter Schwabe
Academia Sinica

ECC2K-130 joint work with:
many, many, many people

What's the best algorithm to attack your favorite cryptosystem? Nobody can really be sure.

For any nontrivial problem P: What's the best algorithm for P? Nobody can really be sure.

What's the best algorithm to attack your favorite cryptosystem? Nobody can really be sure.

For any nontrivial problem P: What's the best algorithm for P? Nobody can really be sure.

But can *estimate* the cost of this algorithm as the cost of the best algorithm known.

What's the best algorithm to attack your favorite cryptosystem? Nobody can really be sure.

For any nontrivial problem P: What's the best algorithm for P? Nobody can really be sure.

But can *estimate* the cost of this algorithm as the cost of the best algorithm known.

Does this estimate inspire confidence? Maybe, maybe not!

How precise is the estimate?

Compare “exponential in n”
to “$(1.1 + o(1))^n$” to “$n^{O(1)} 1.1^n$”
to “$37n^2 1.1^n$ bit operations.”

How precise is the estimate?

Compare “exponential in n” to “$(1.1 + o(1))^n$” to “$n^{O(1)}1.1^n$” to “$37n^2 1.1^n$ bit operations.”

How slowly is it changing?

Consider matrix-mult exponent:

2.81 (1969). 2.796 (1978).

2.78 (1979). 2.522 (1981).

2.517 (1982). 2.496 (1981).

2.479 (1986). 2.376 (1989).

How precise is the estimate?

Compare "exponential in n" to "$(1.1 + o(1))^n$" to "$n^{O(1)} 1.1^n$" to "$37n^2 1.1^n$ bit operations."

How slowly is it changing?

Consider matrix-mult exponent:

2.81 (1969). 2.796 (1978).
2.78 (1979). 2.522 (1981).
2.517 (1982). 2.496 (1981).
2.479 (1986). 2.376 (1989).
2.374 (2010). 2.373 (2011).

How precise is the estimate?
Compare "exponential in n"
to "$(1.1 + o(1))^n$" to "$n^{O(1)} 1.1^n$"
to "$37n^2 1.1^n$ bit operations."

How slowly is it changing?
Consider matrix-mult exponent:
2.81 (1969). 2.796 (1978).
2.78 (1979). 2.522 (1981).
2.517 (1982). 2.496 (1981).
2.479 (1986). 2.376 (1989).
2.374 (2010). 2.373 (2011).

How extensive is the literature?
"Look at all these people who
couldn't find better algorithms."

The rho method

Group $\langle P\rangle$ of prime order ℓ.
Discrete-log problem for $\langle P\rangle$:
given P, kP, find k mod ℓ.

Standard attack: parallel rho.

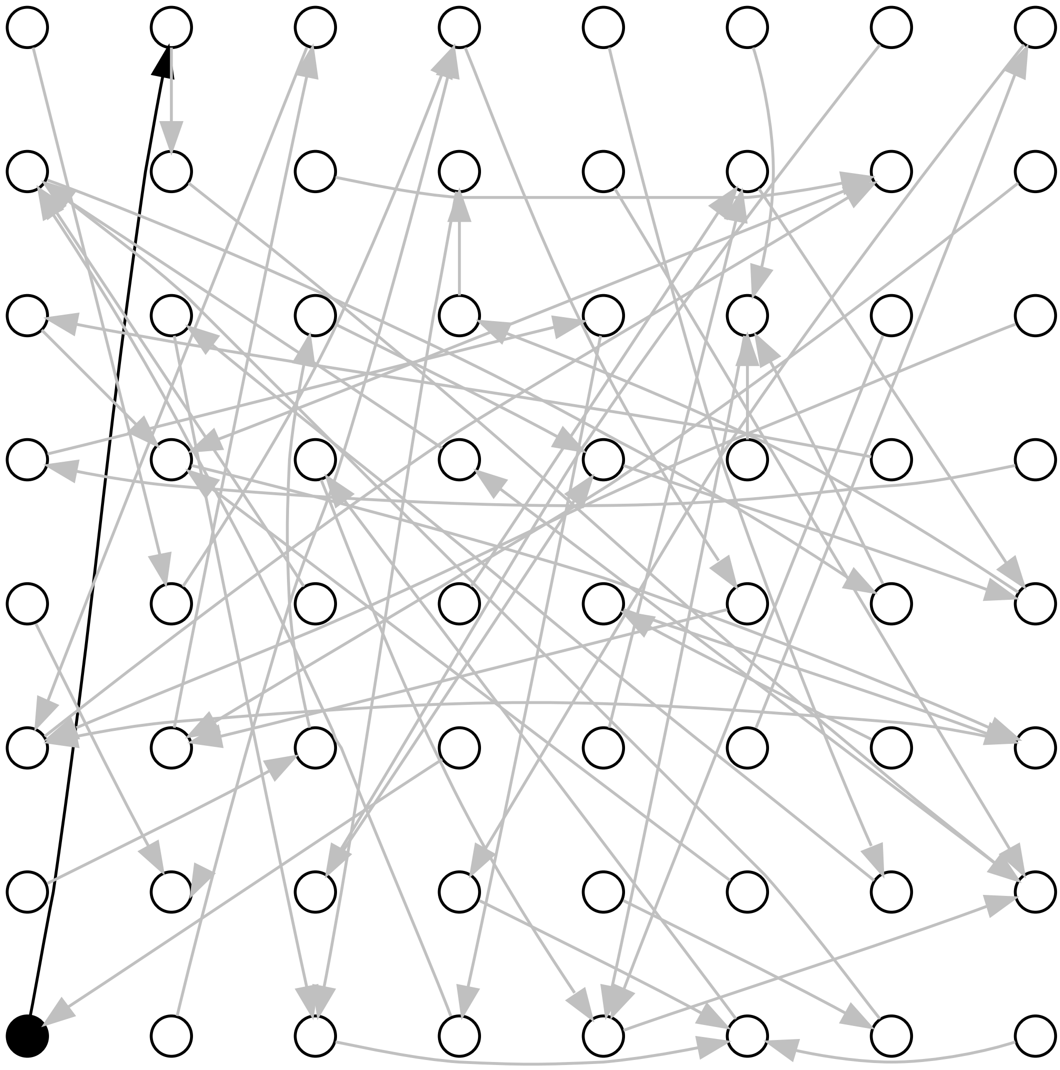
Expect $(1+o(1))\sqrt{\pi\ell/2}$
group operations,
matching lower bound
from Nechaev/Shoup.
Easy to distribute across CPUs.
Very little memory consumption.
Very little communication.

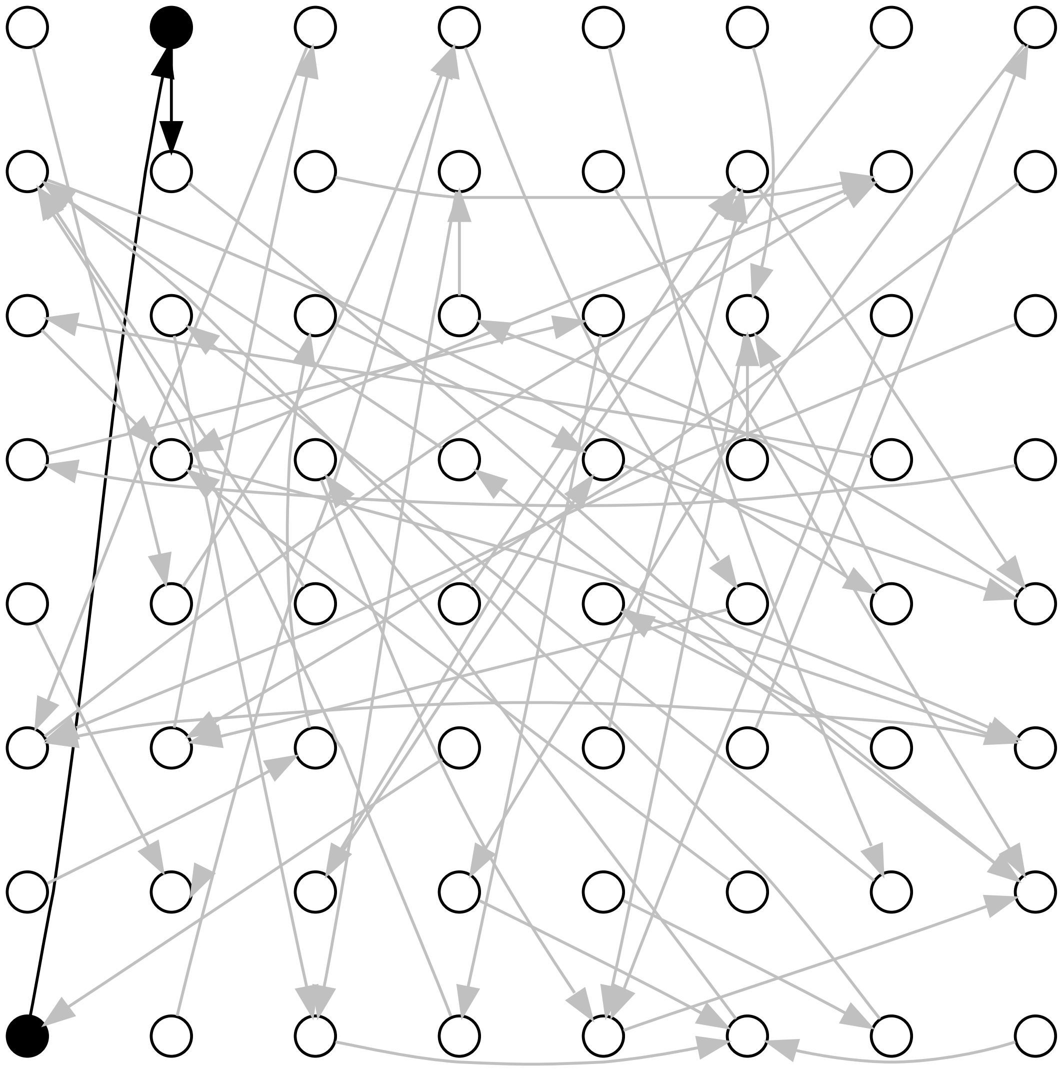
Simplified, non-parallel rho:

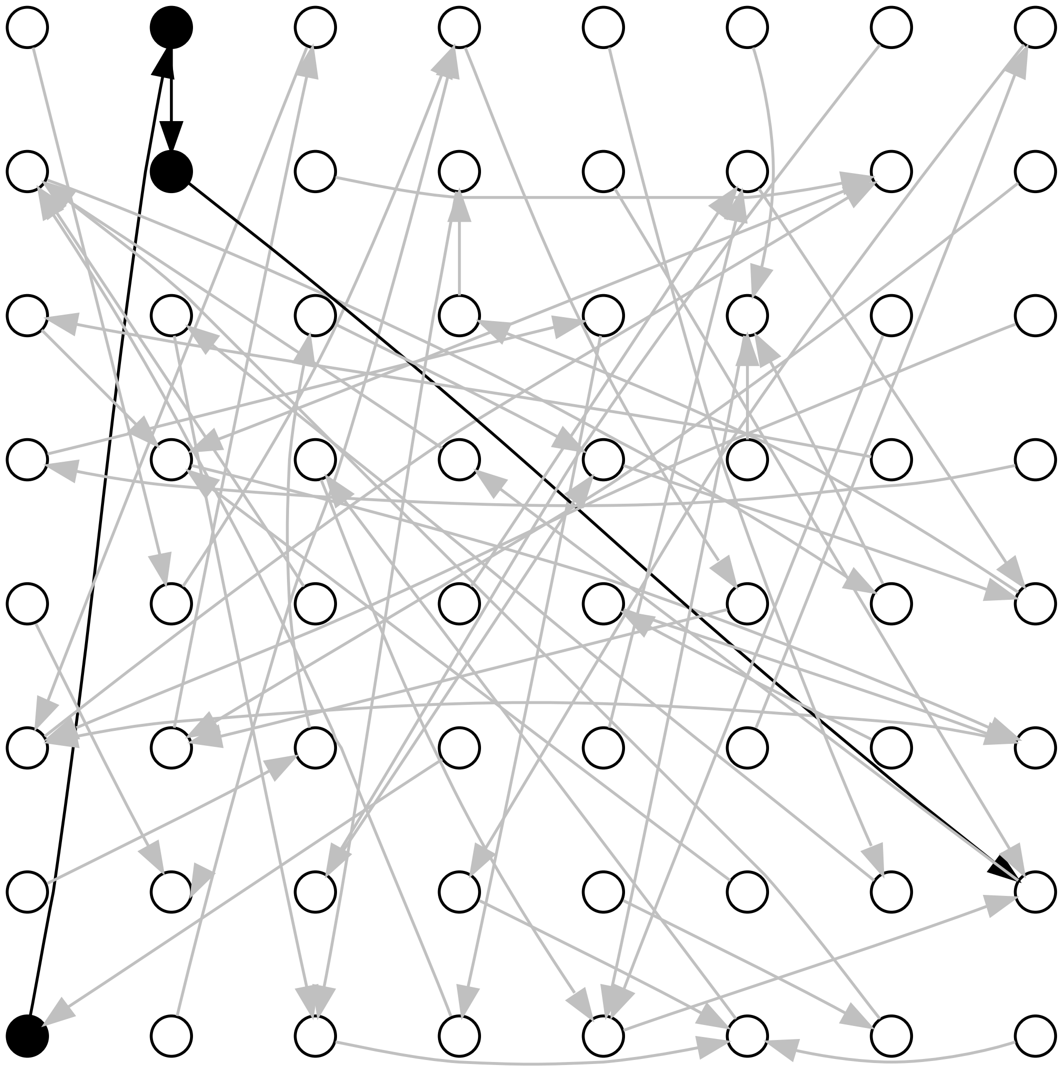
Make a pseudo-random walk in the group $\langle P \rangle$, where the next step depends on current point: $W_{i+1} = f(W_i)$.

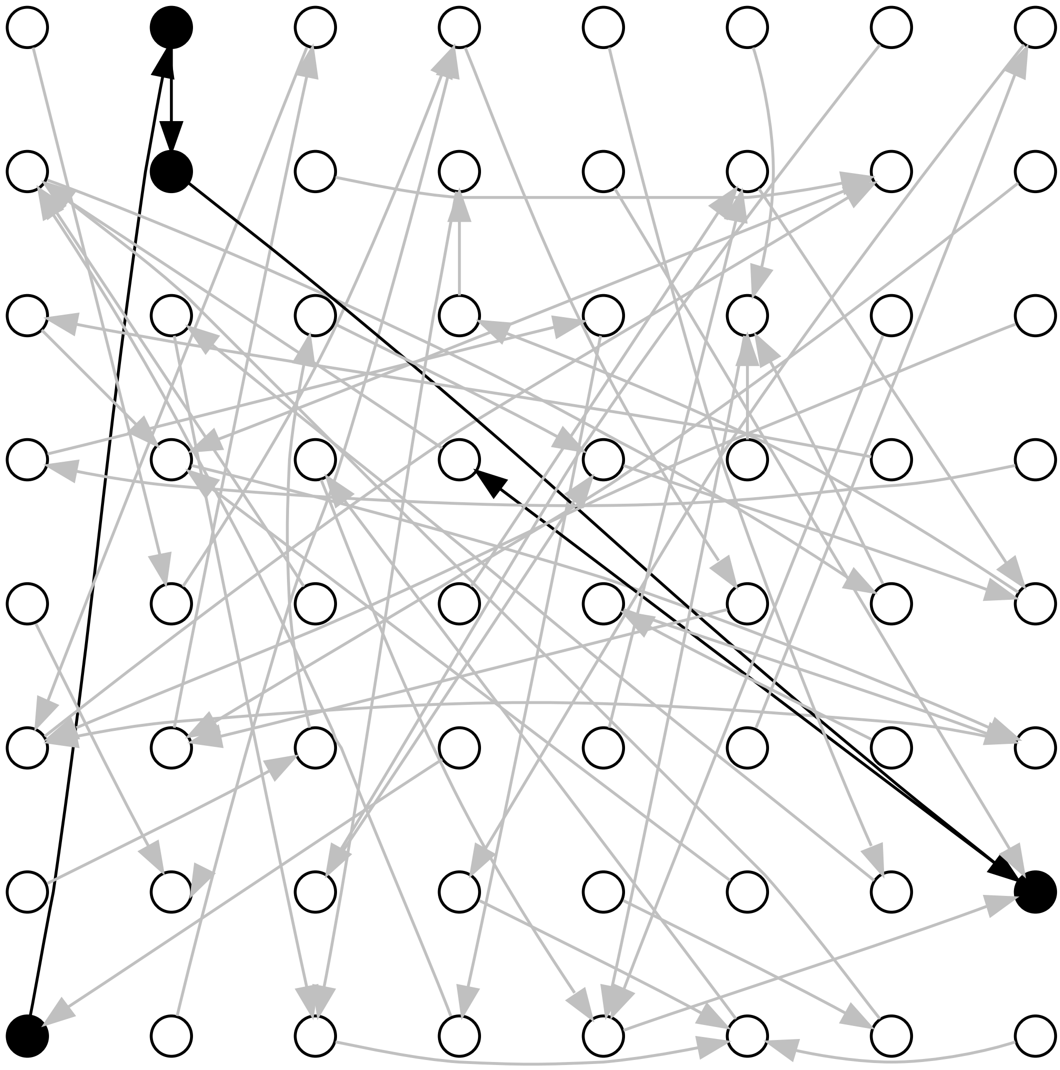
Birthday paradox: Randomly choosing from ℓ elements picks one element twice after about $\sqrt{\pi\ell/2}$ draws.

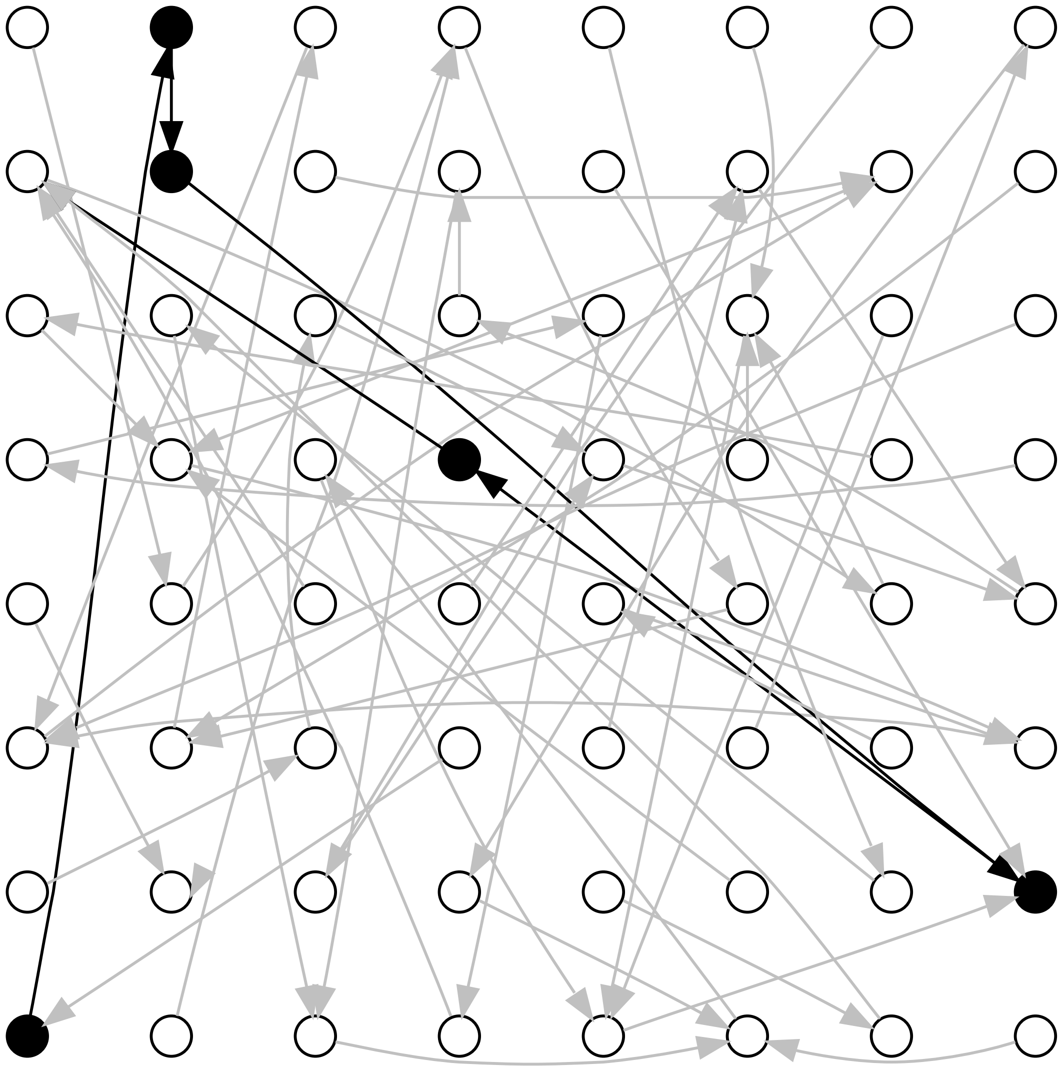
The walk now enters a cycle. Cycle-finding algorithm (e.g., Floyd) quickly detects this.

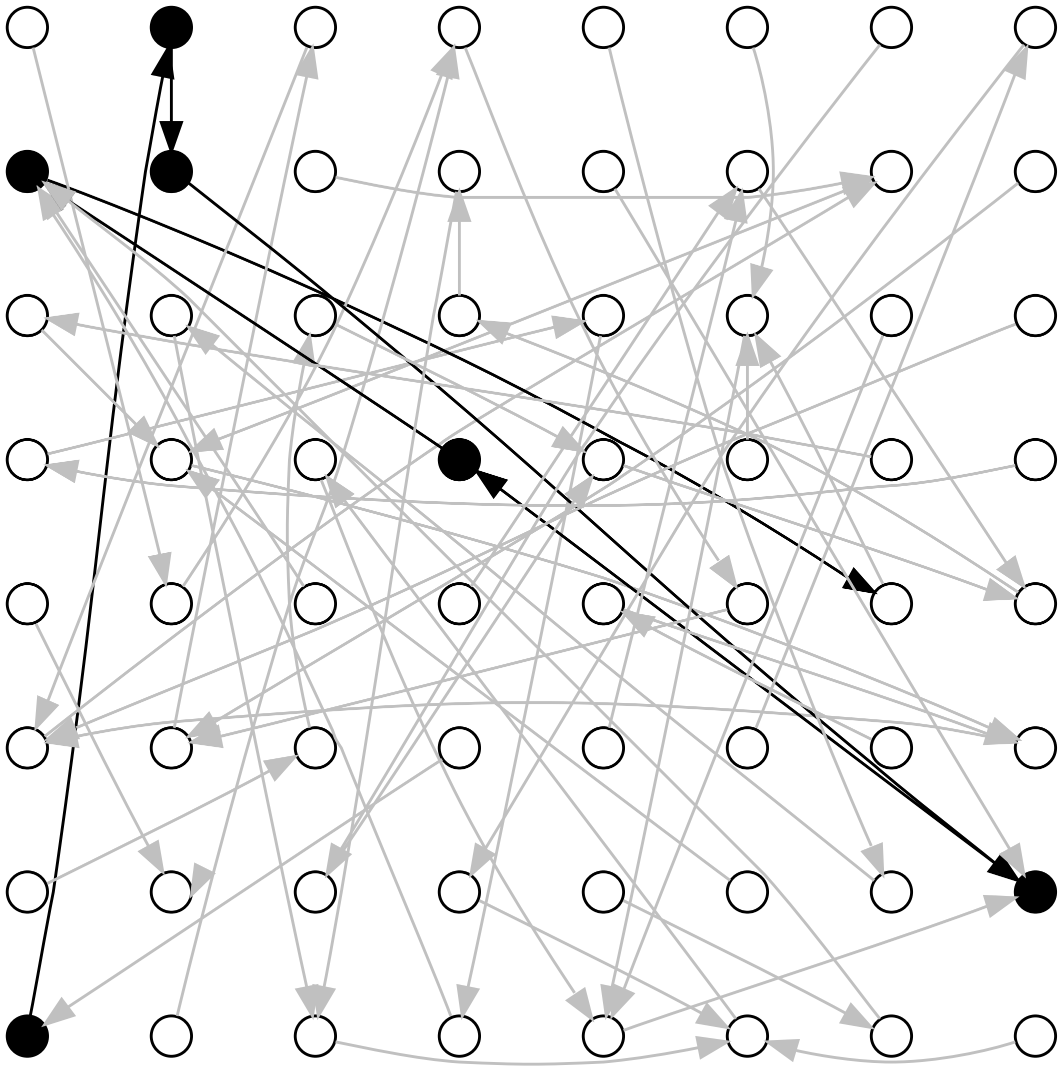


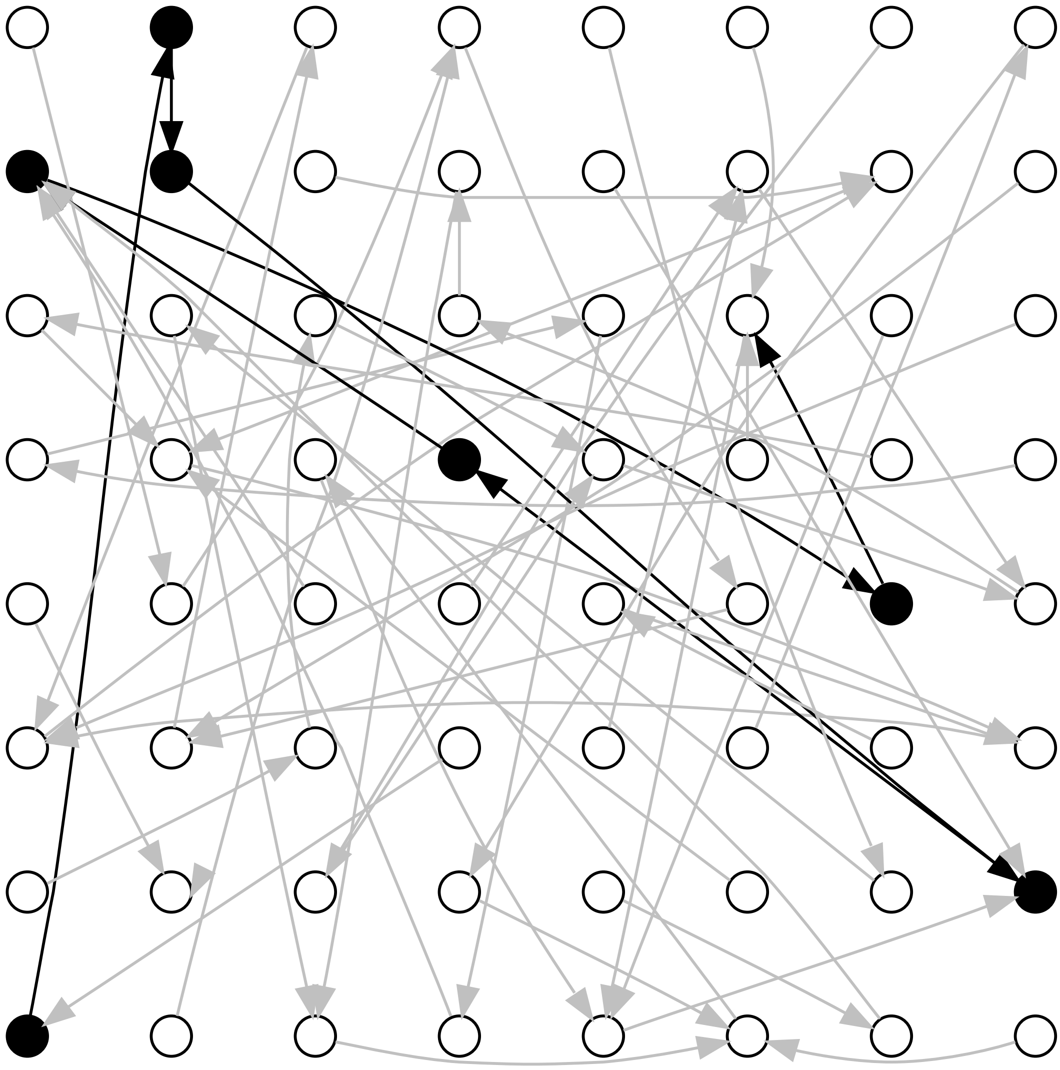


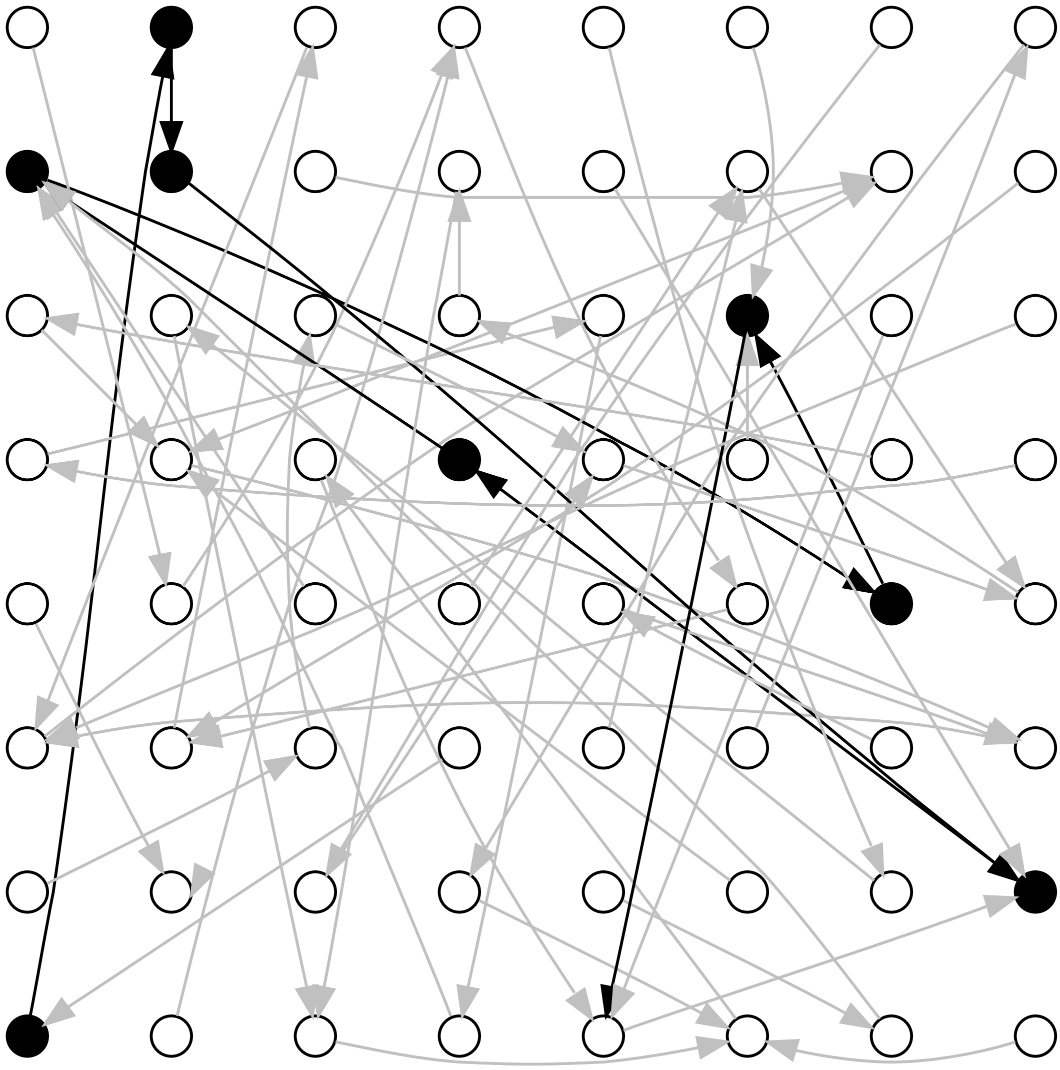


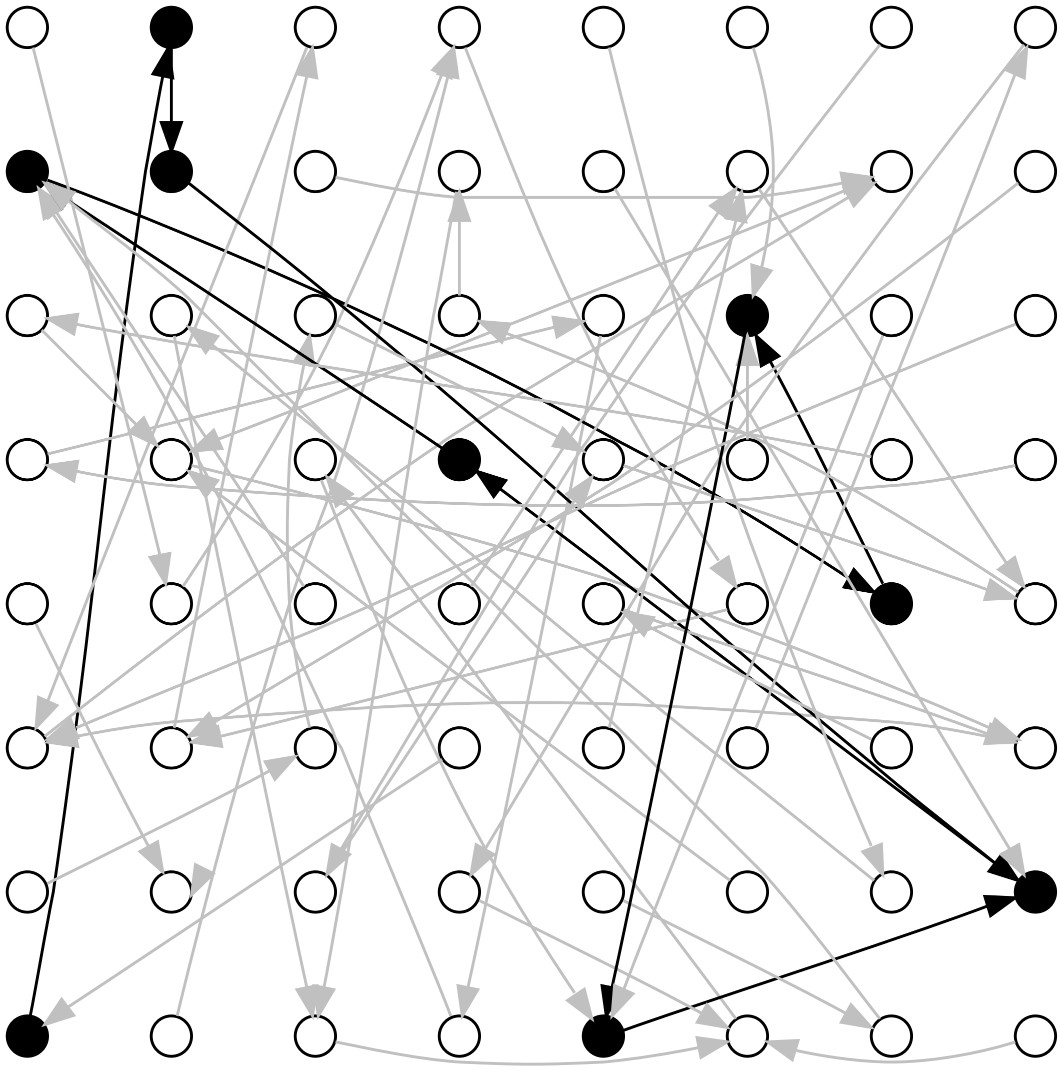


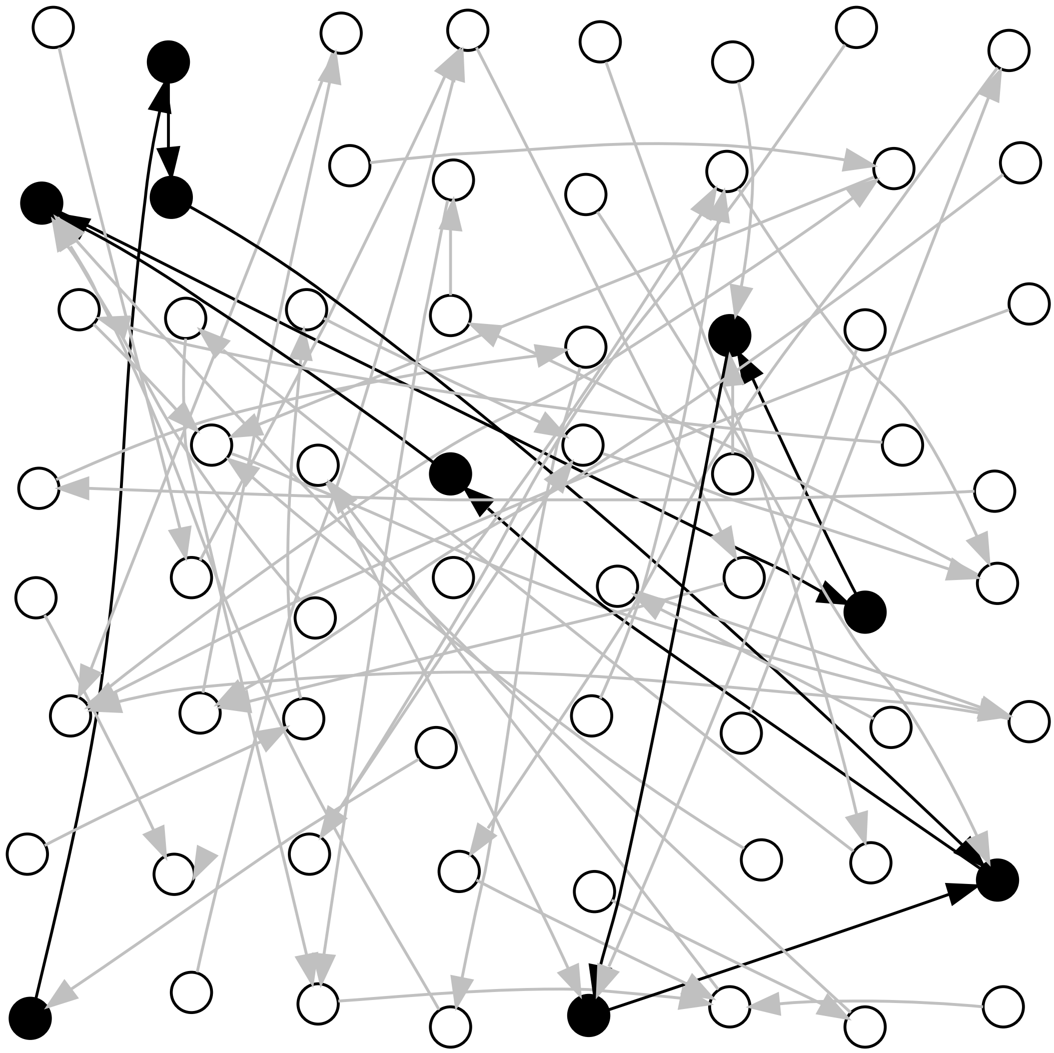


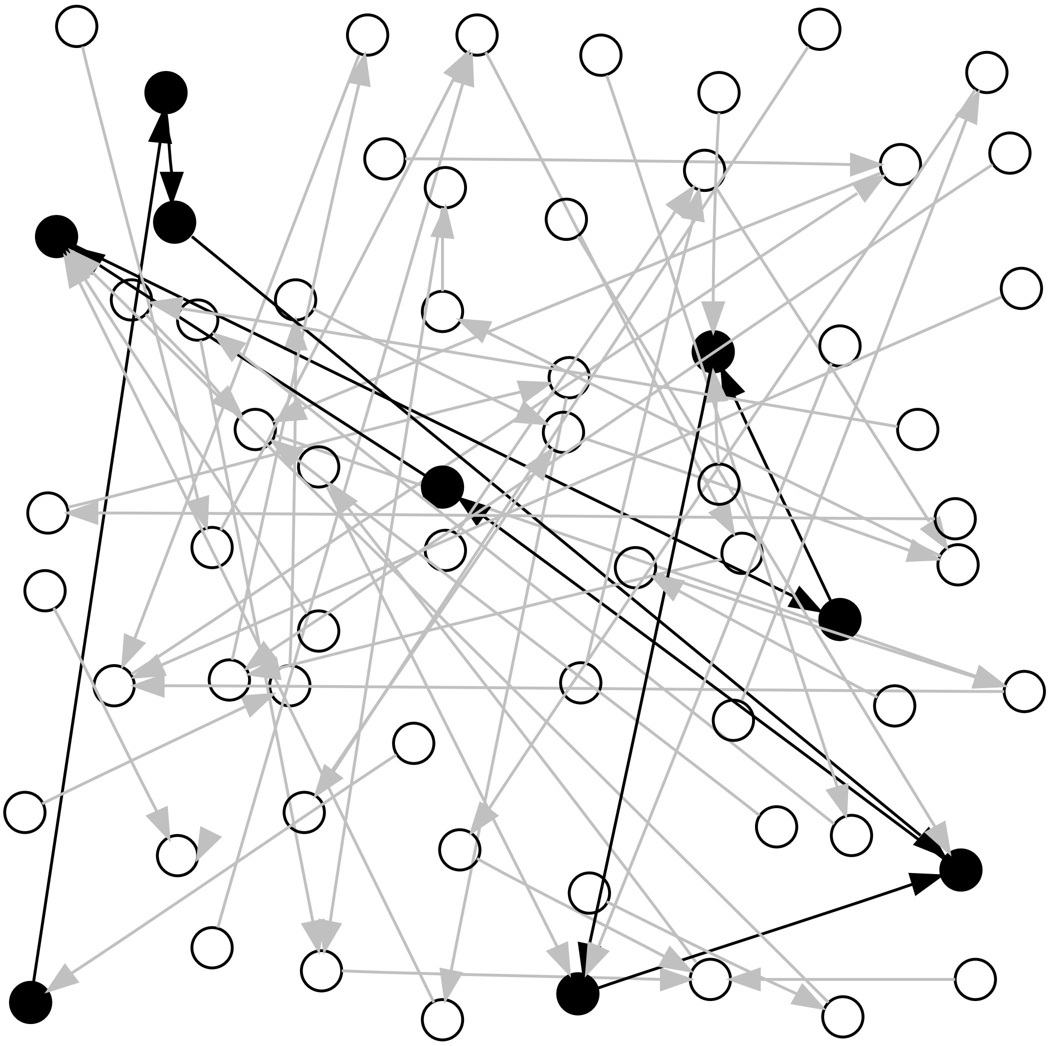


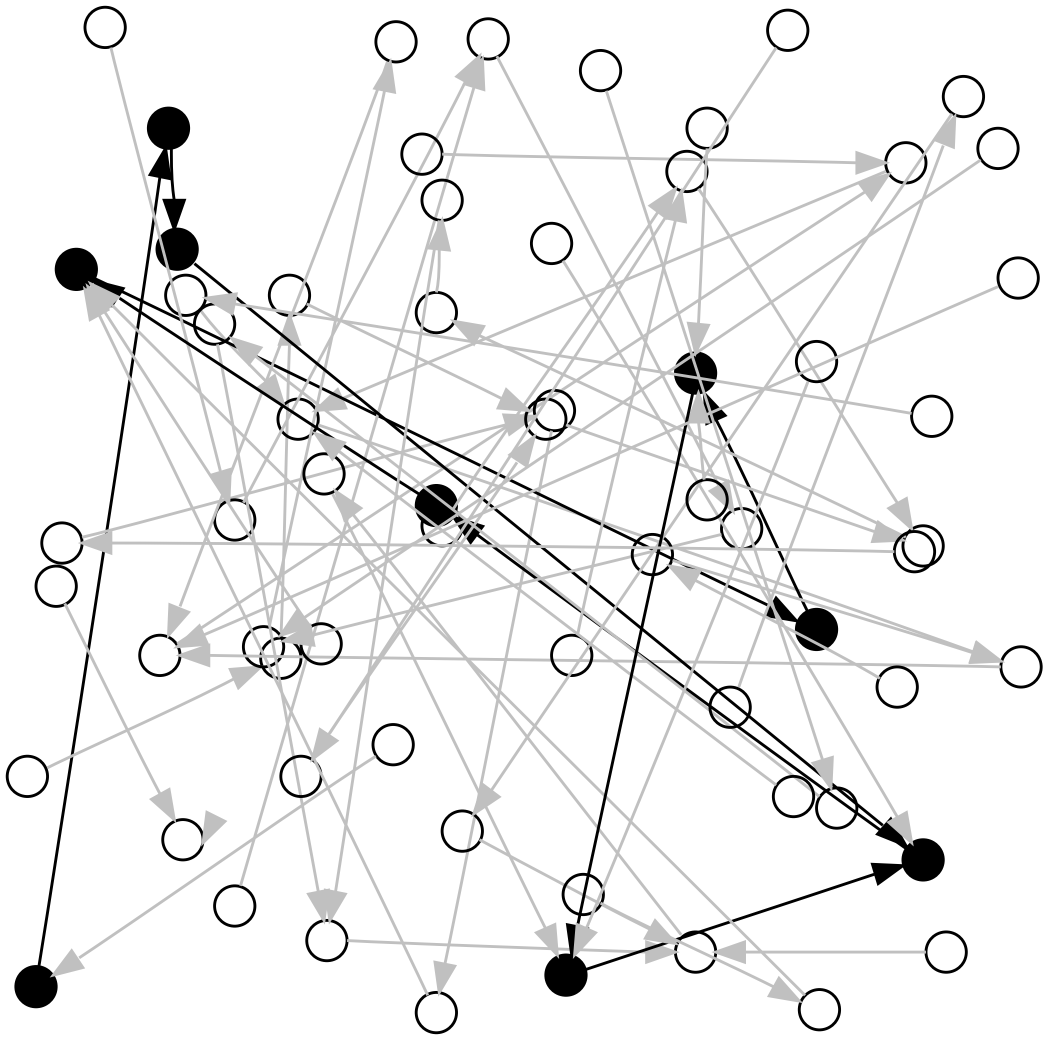


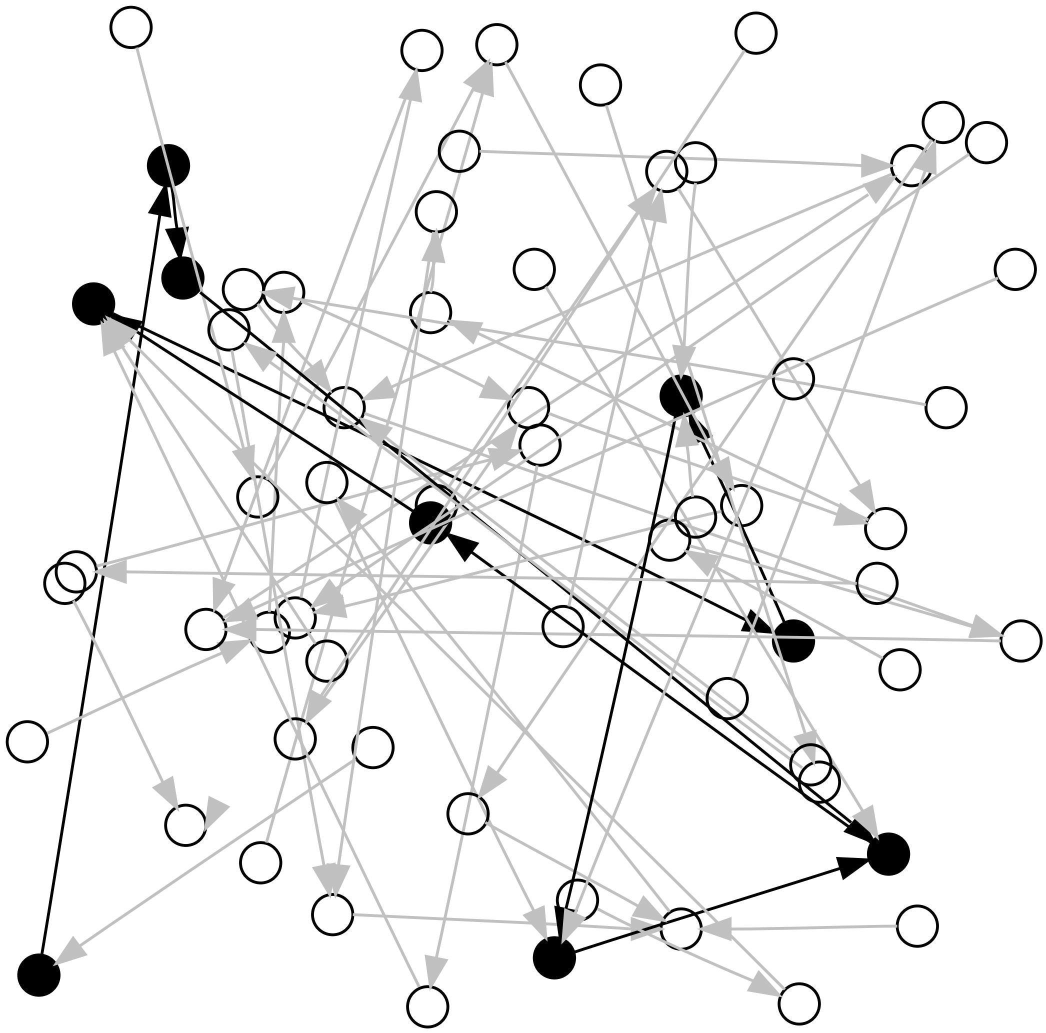


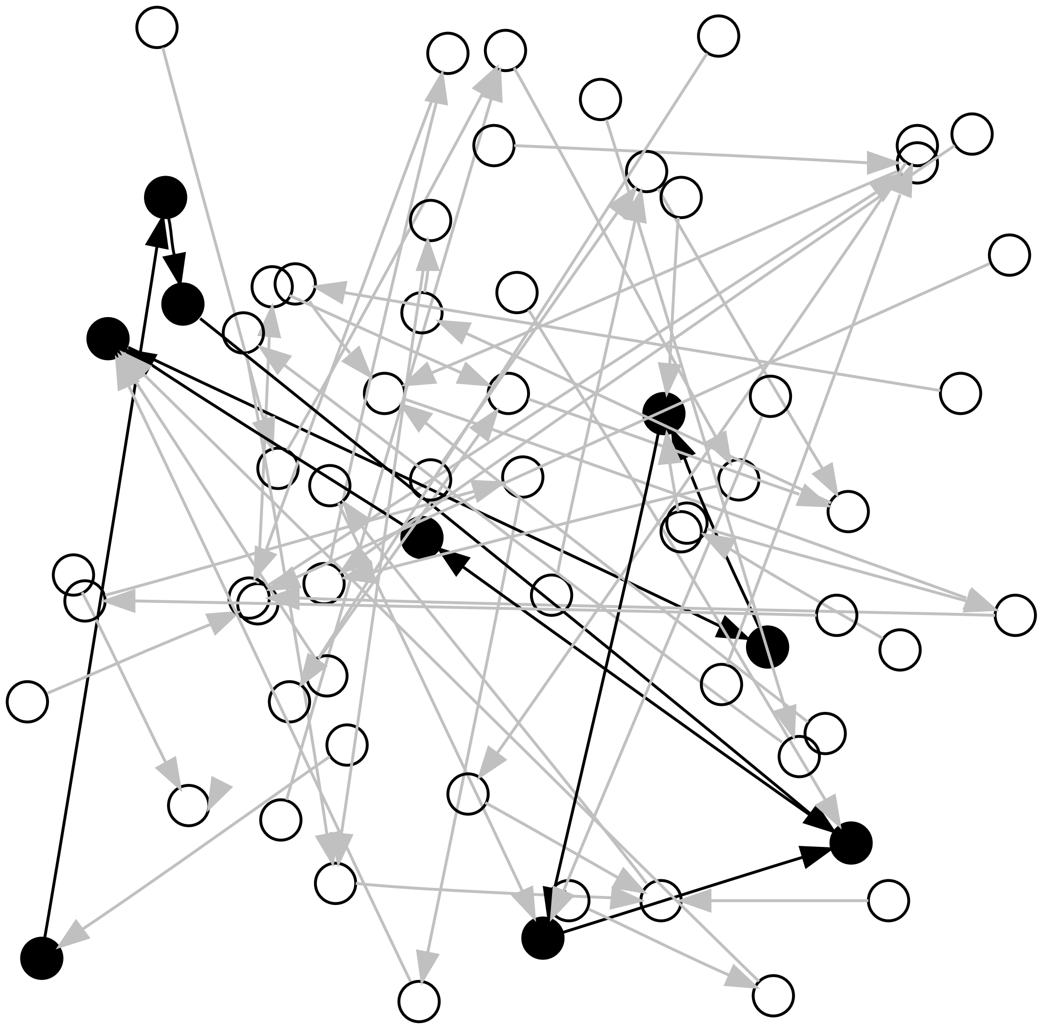


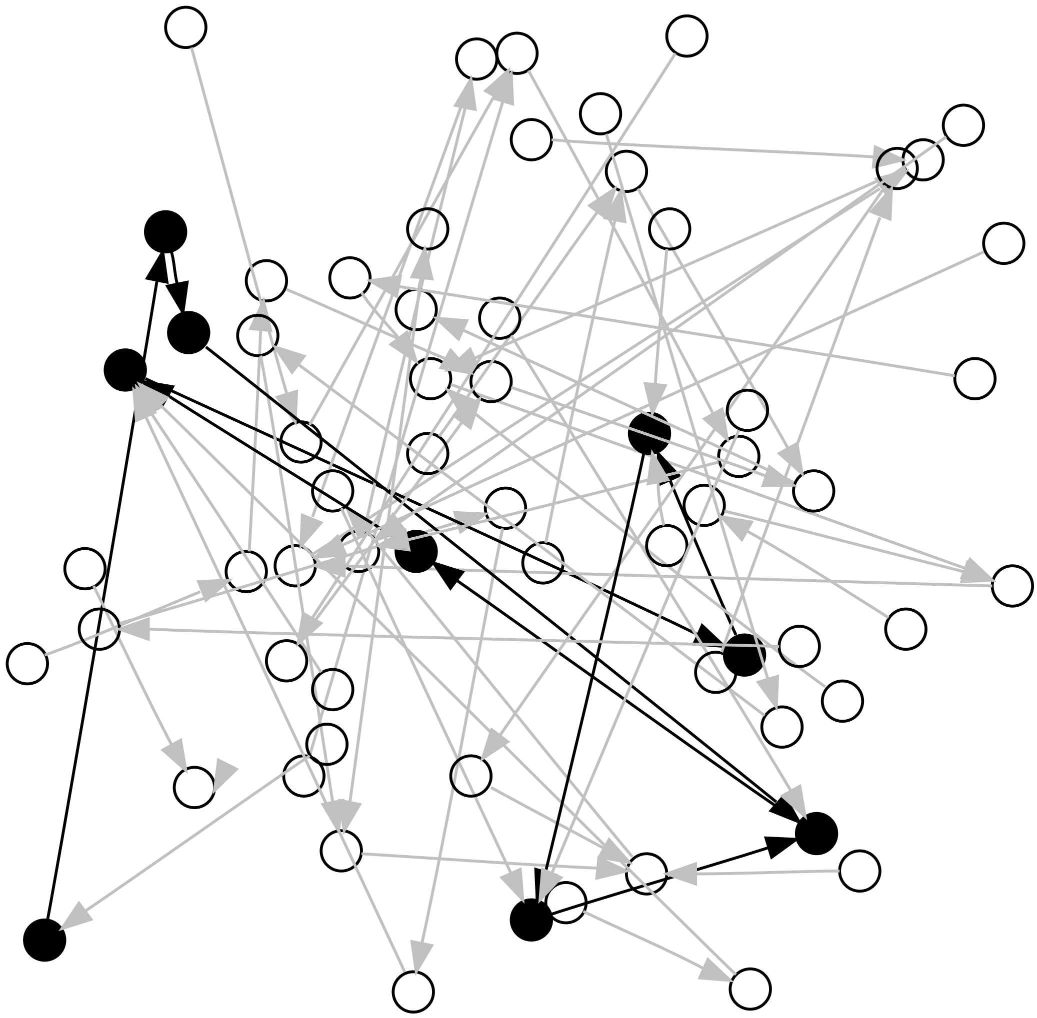


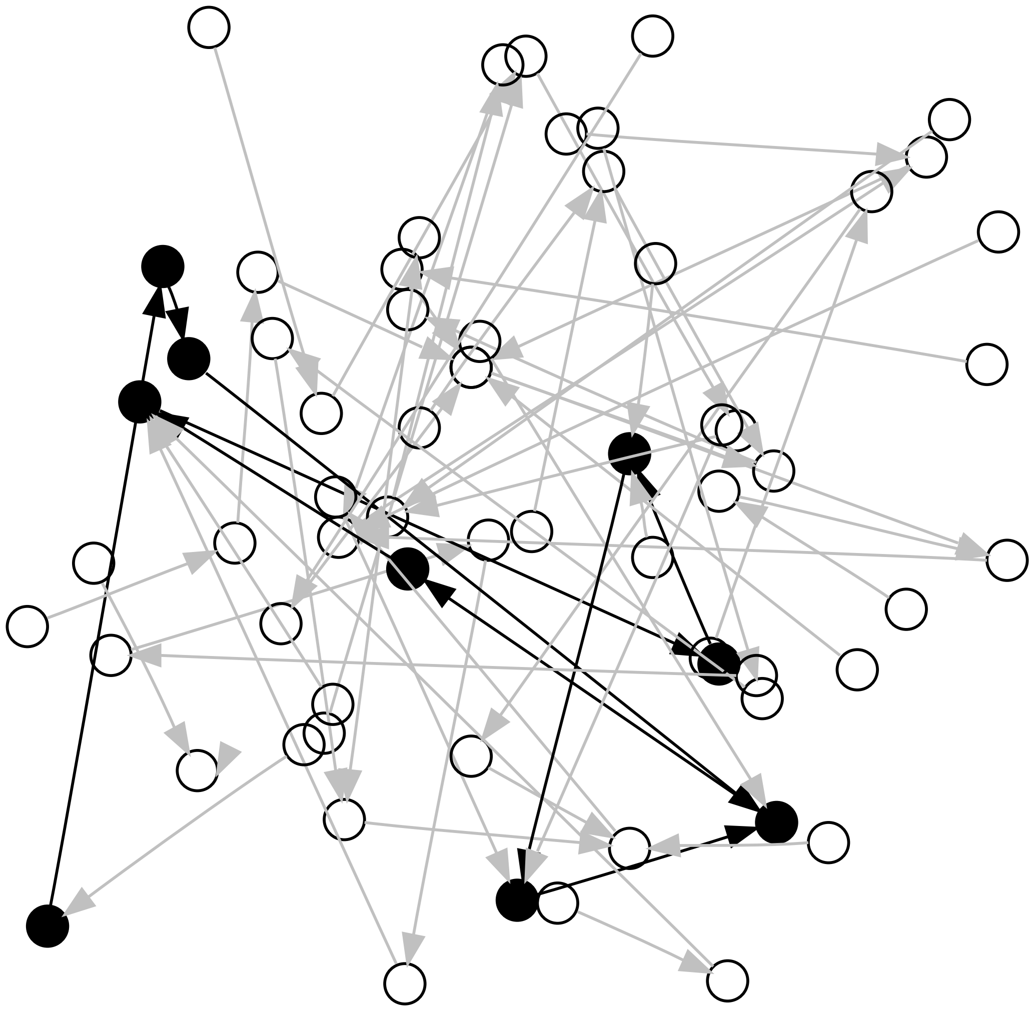


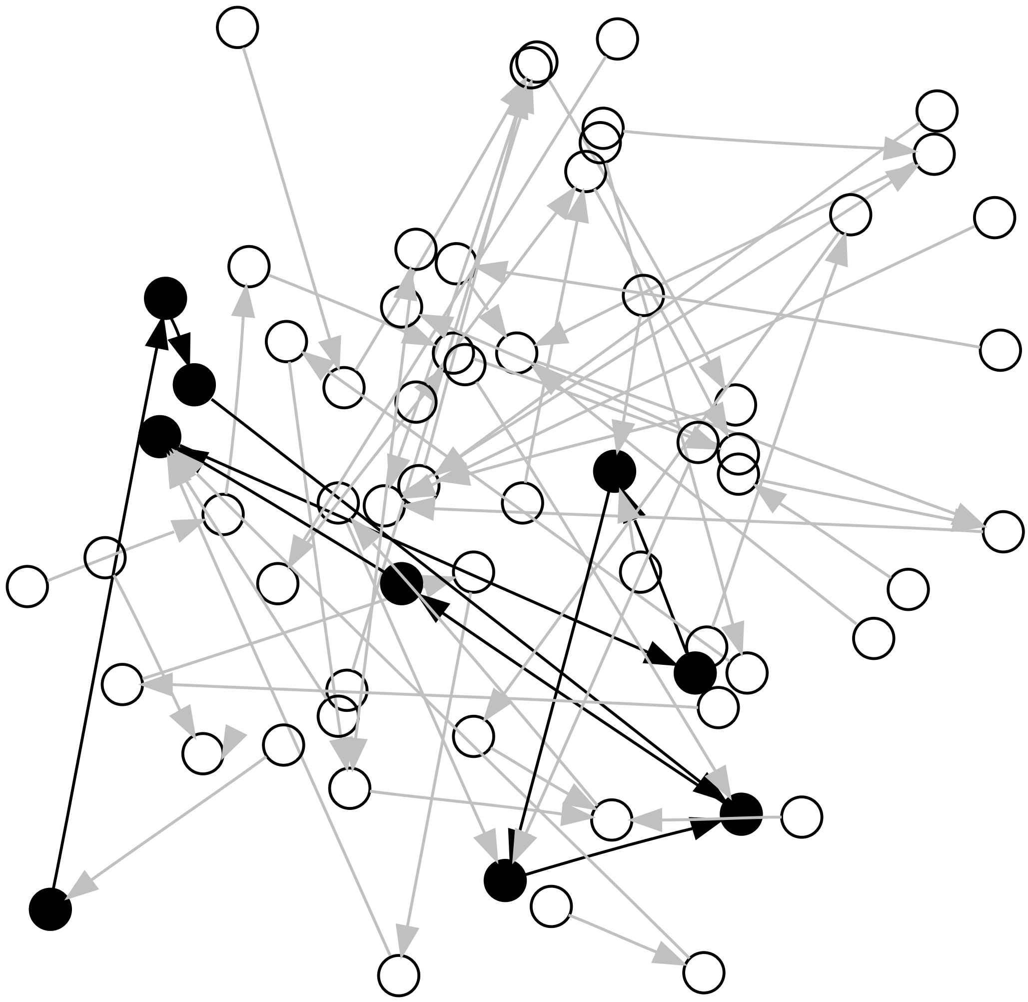


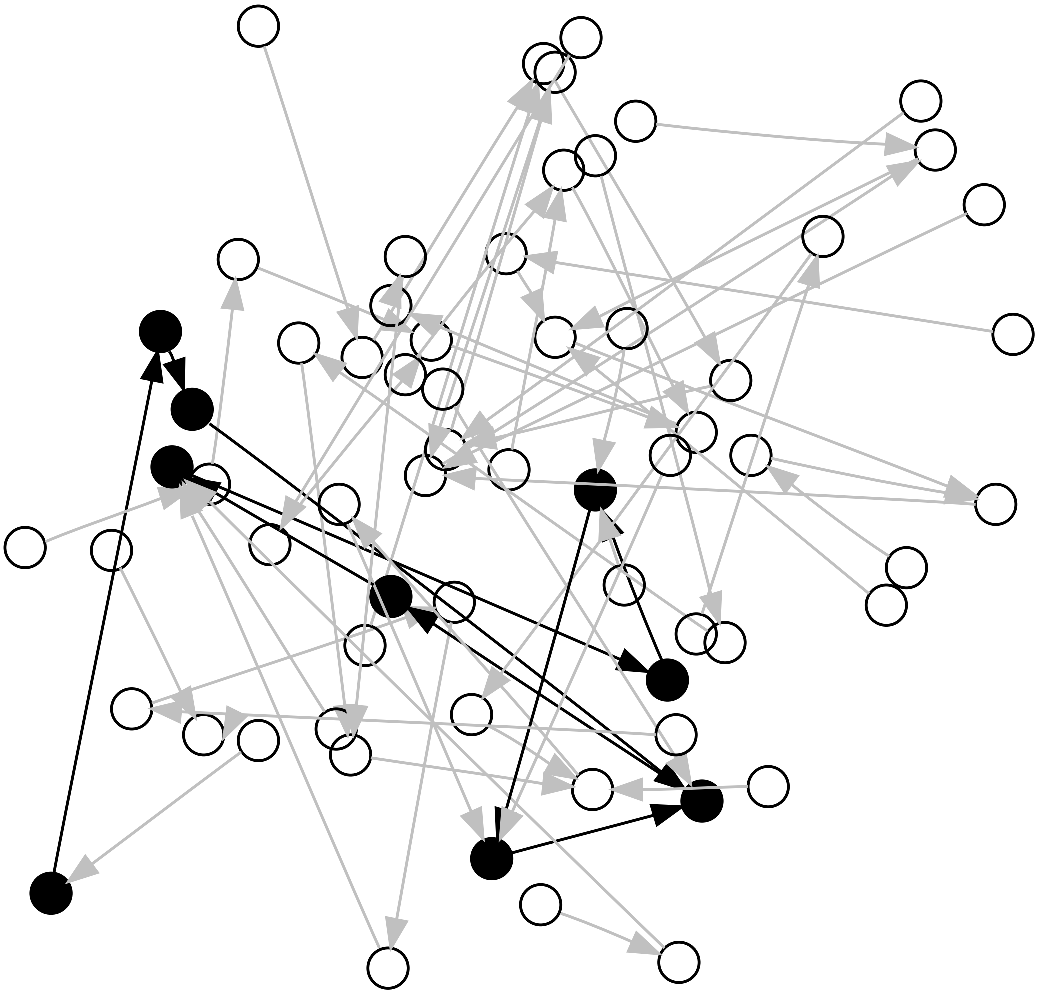


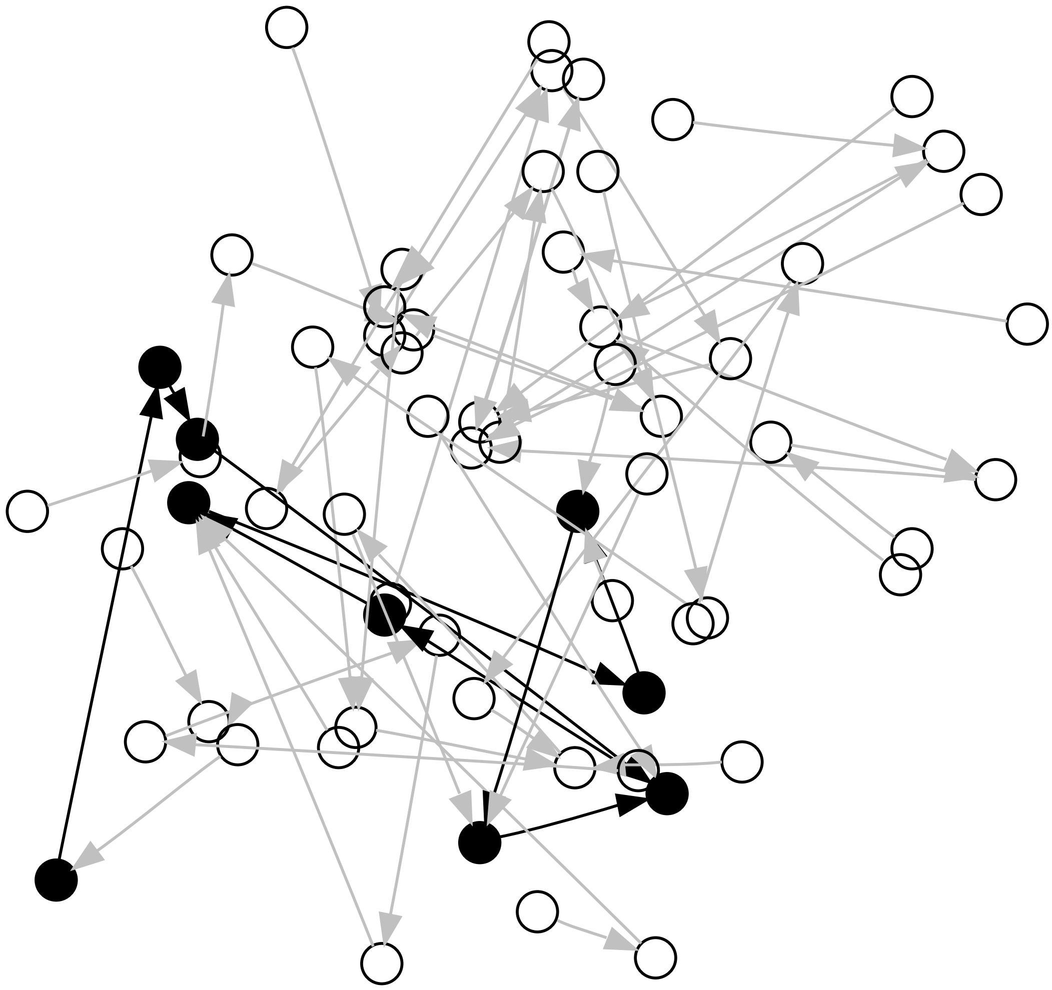


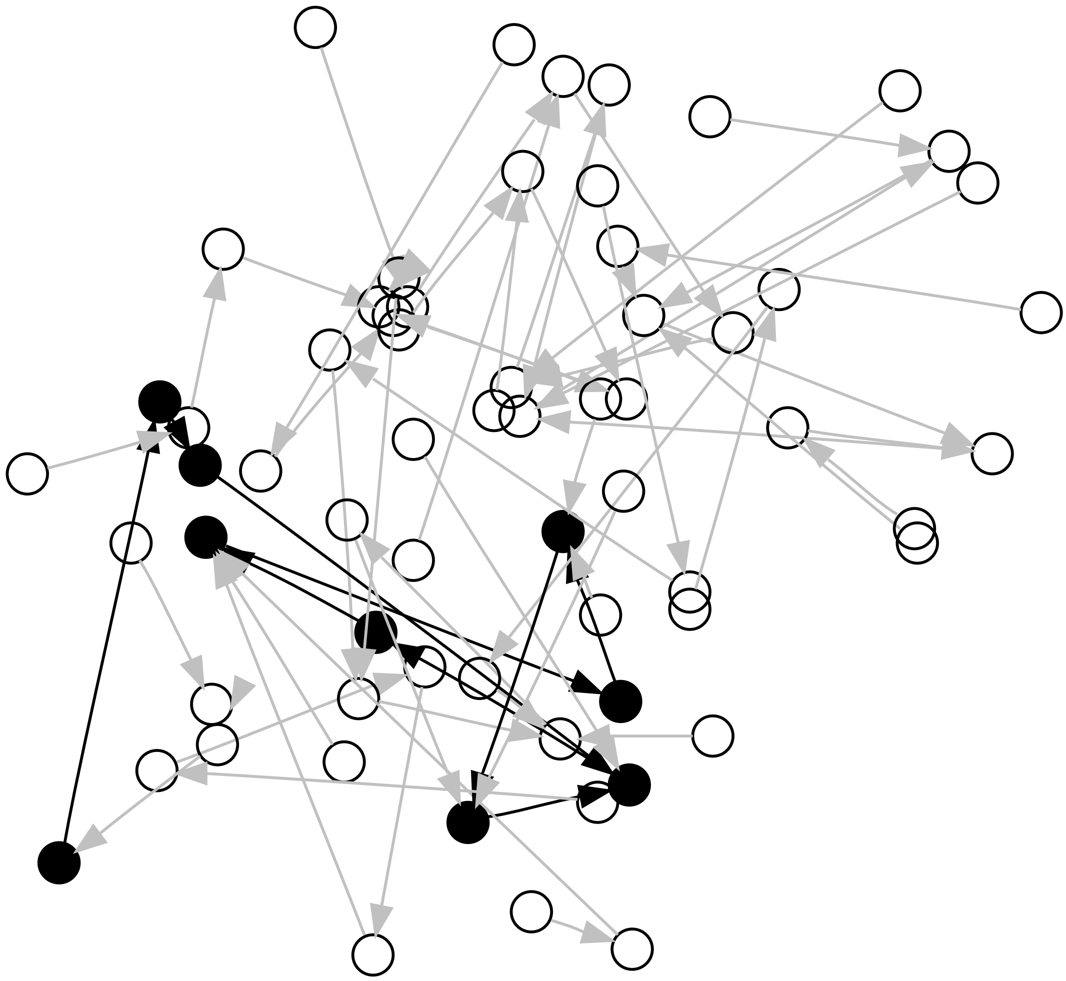


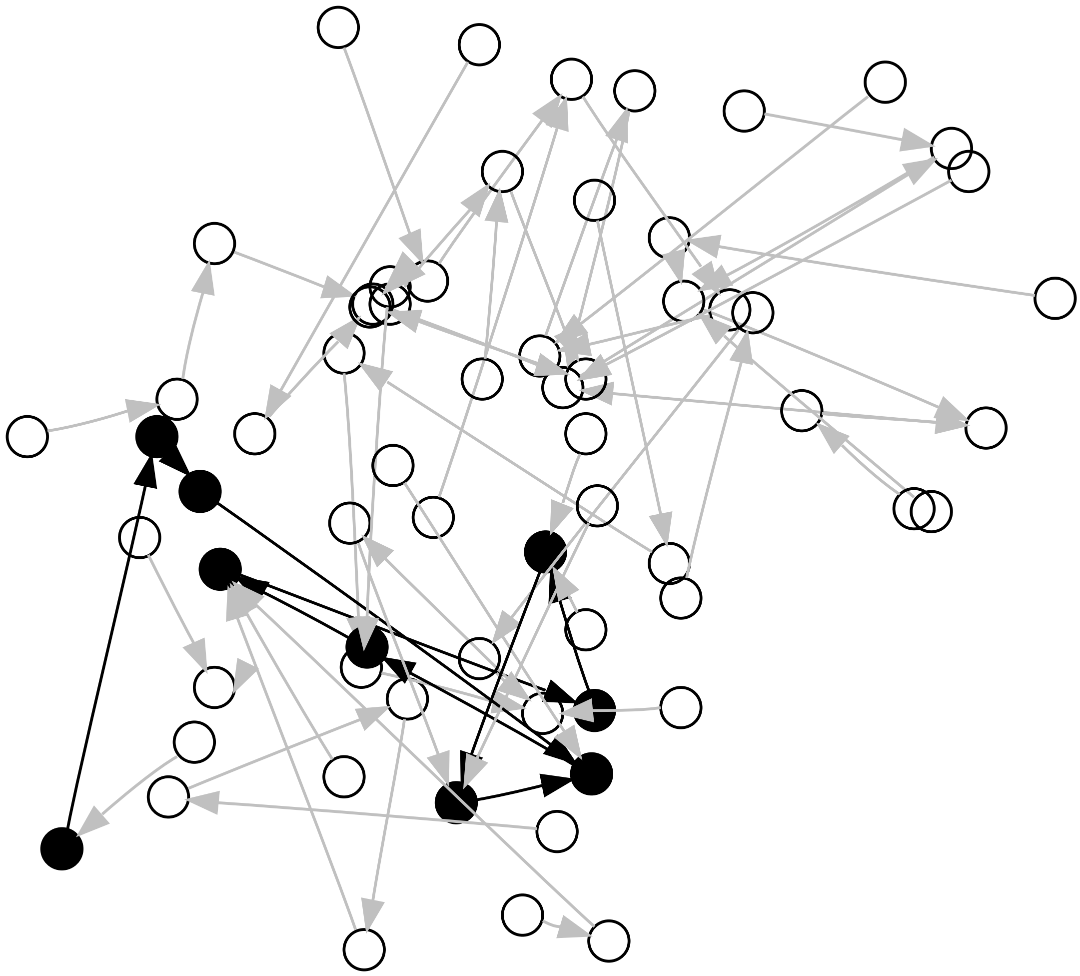


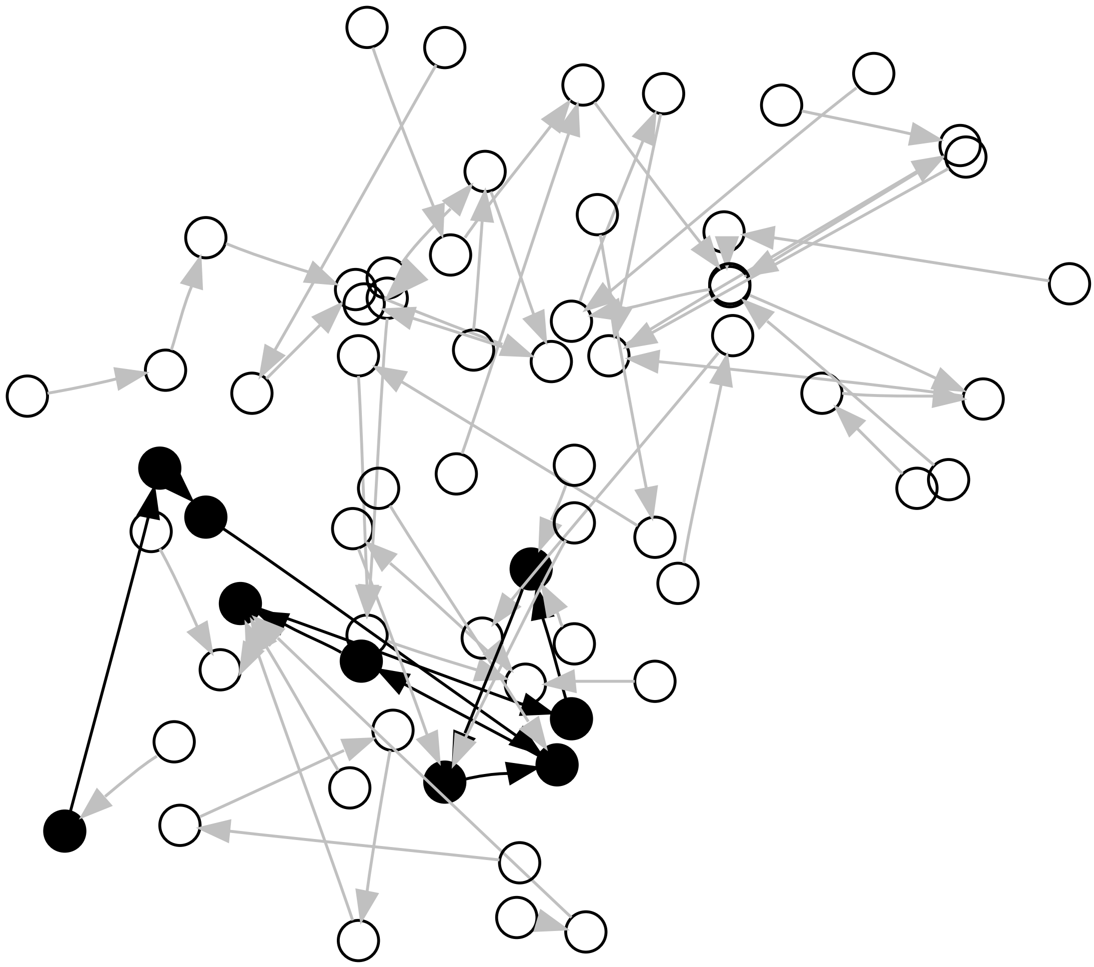


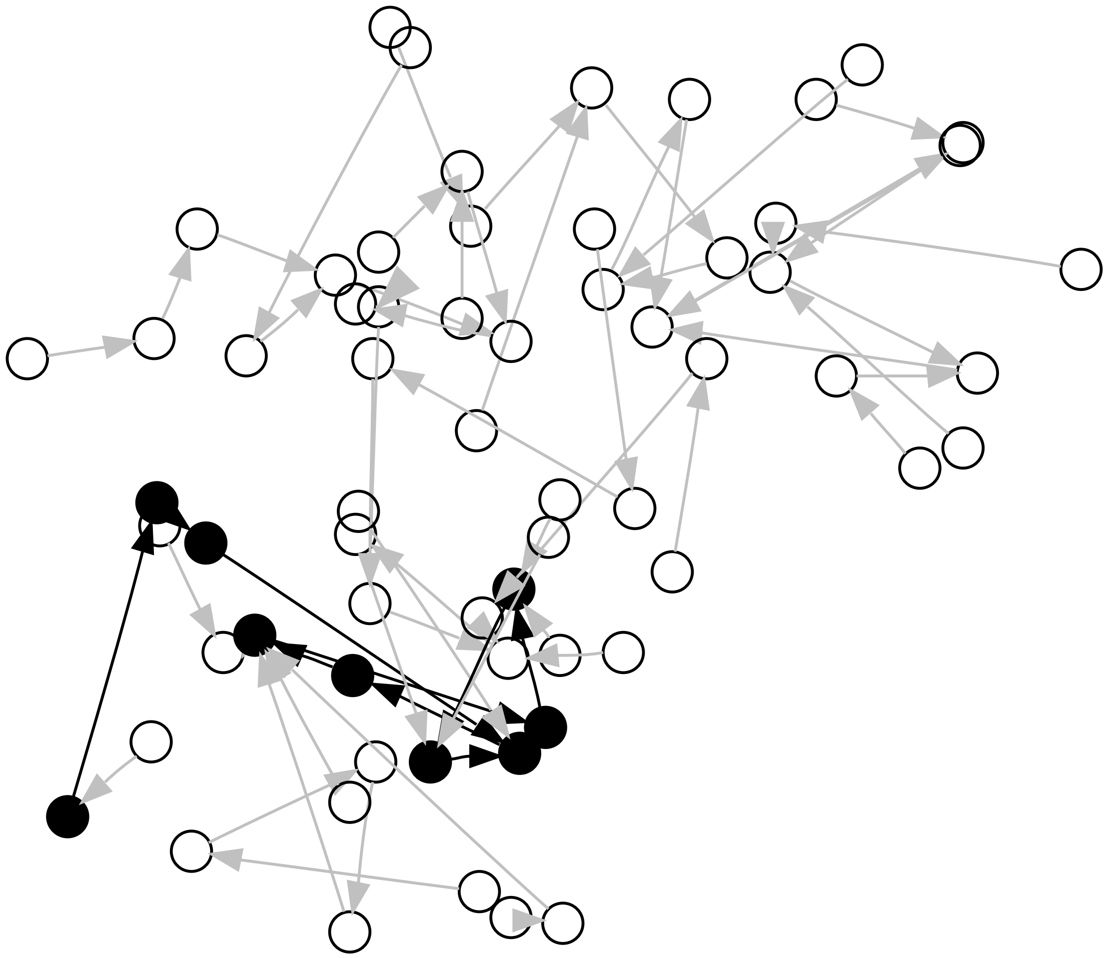


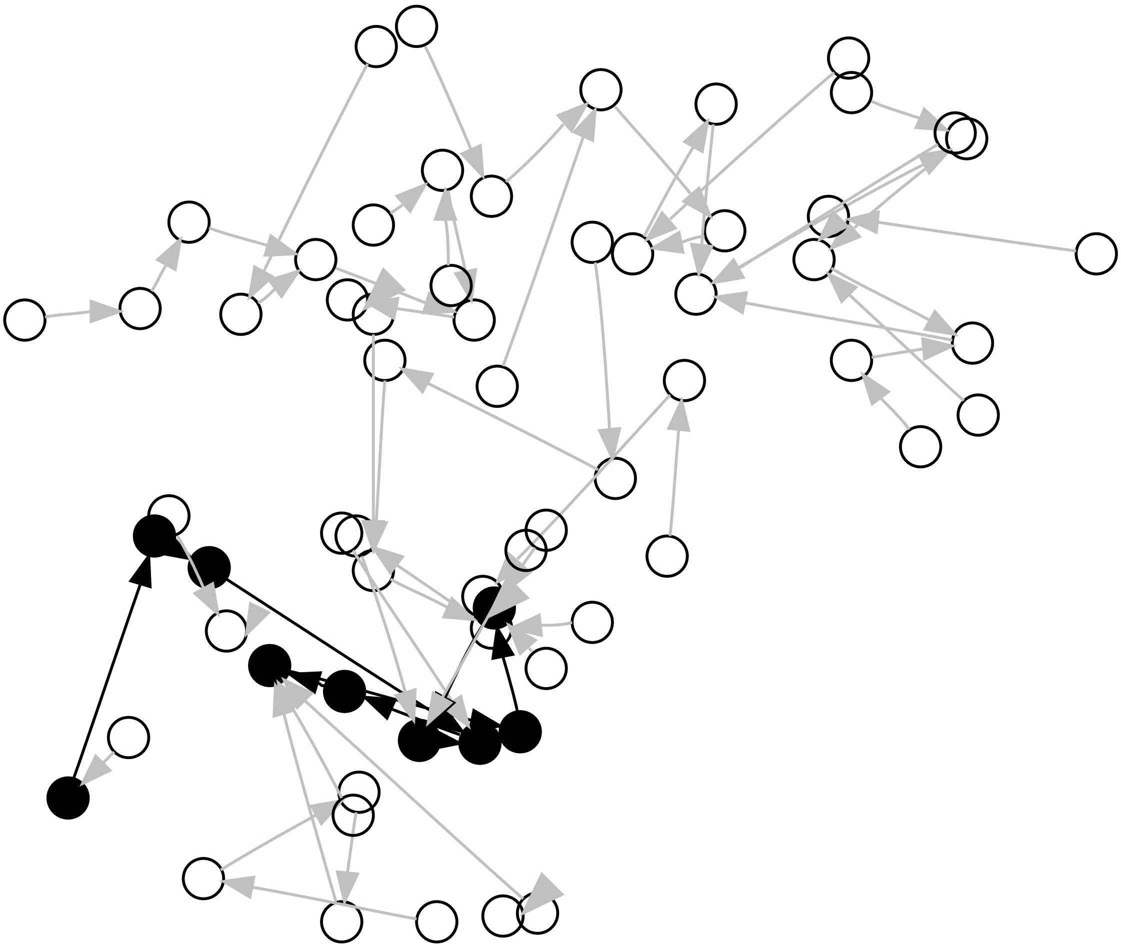


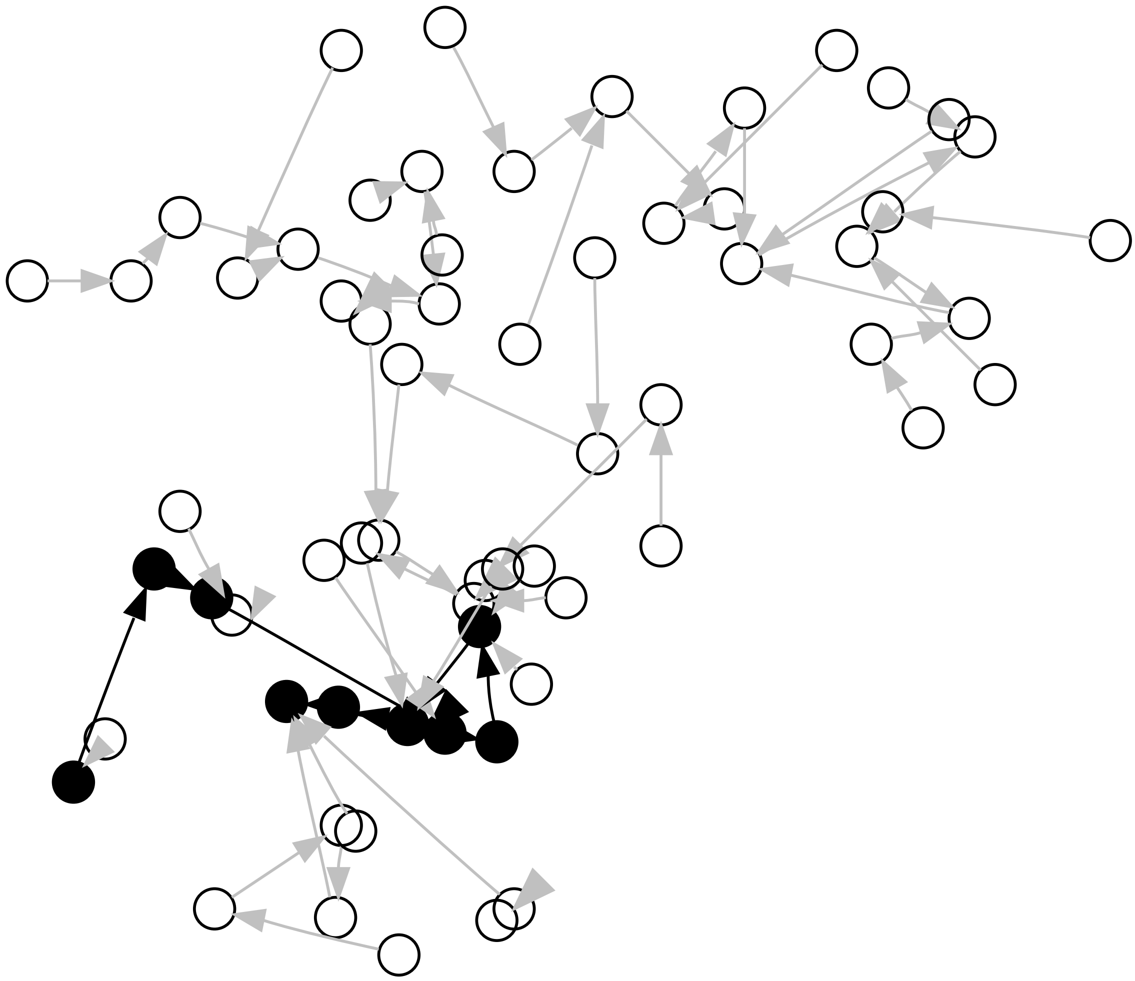


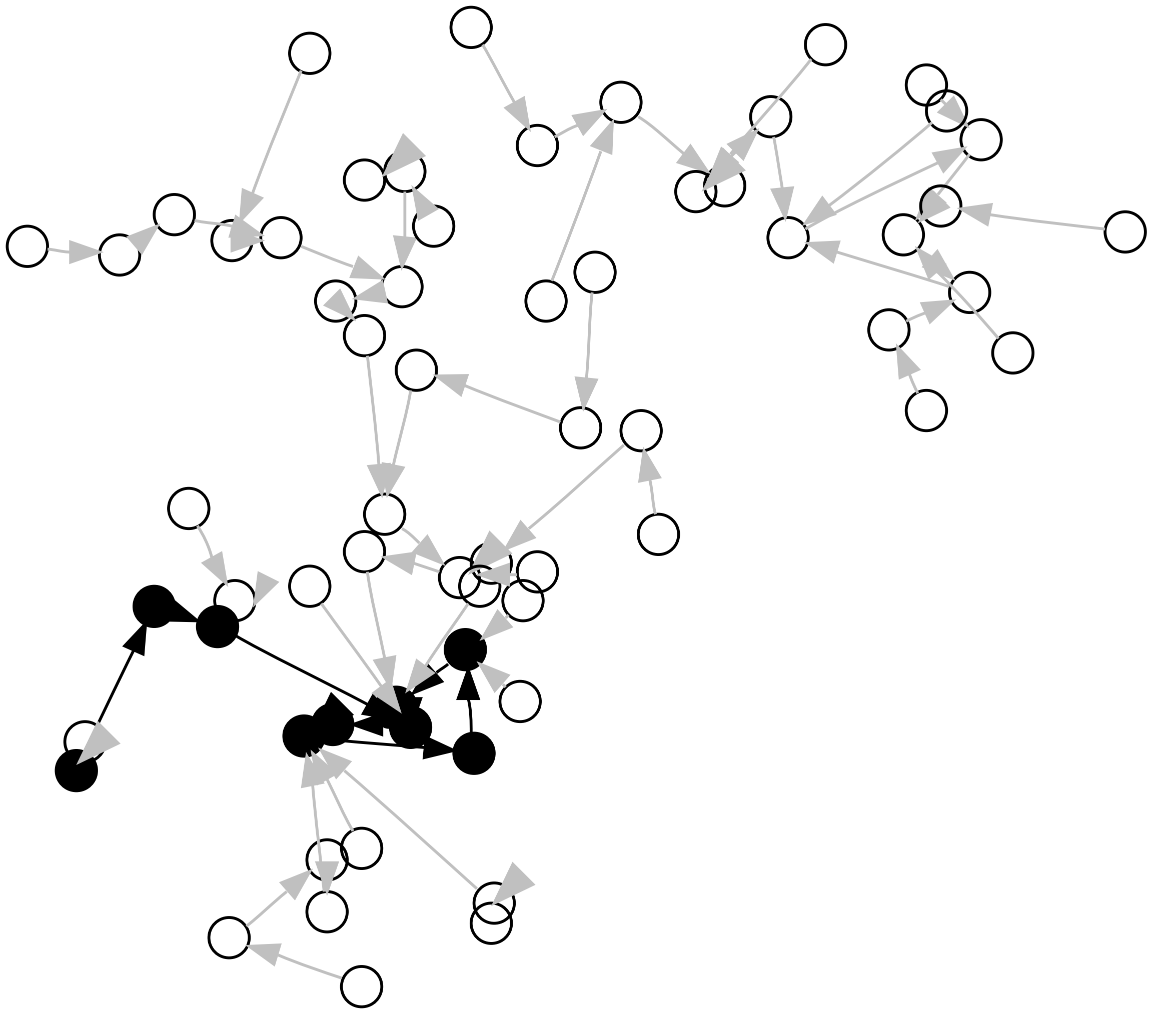


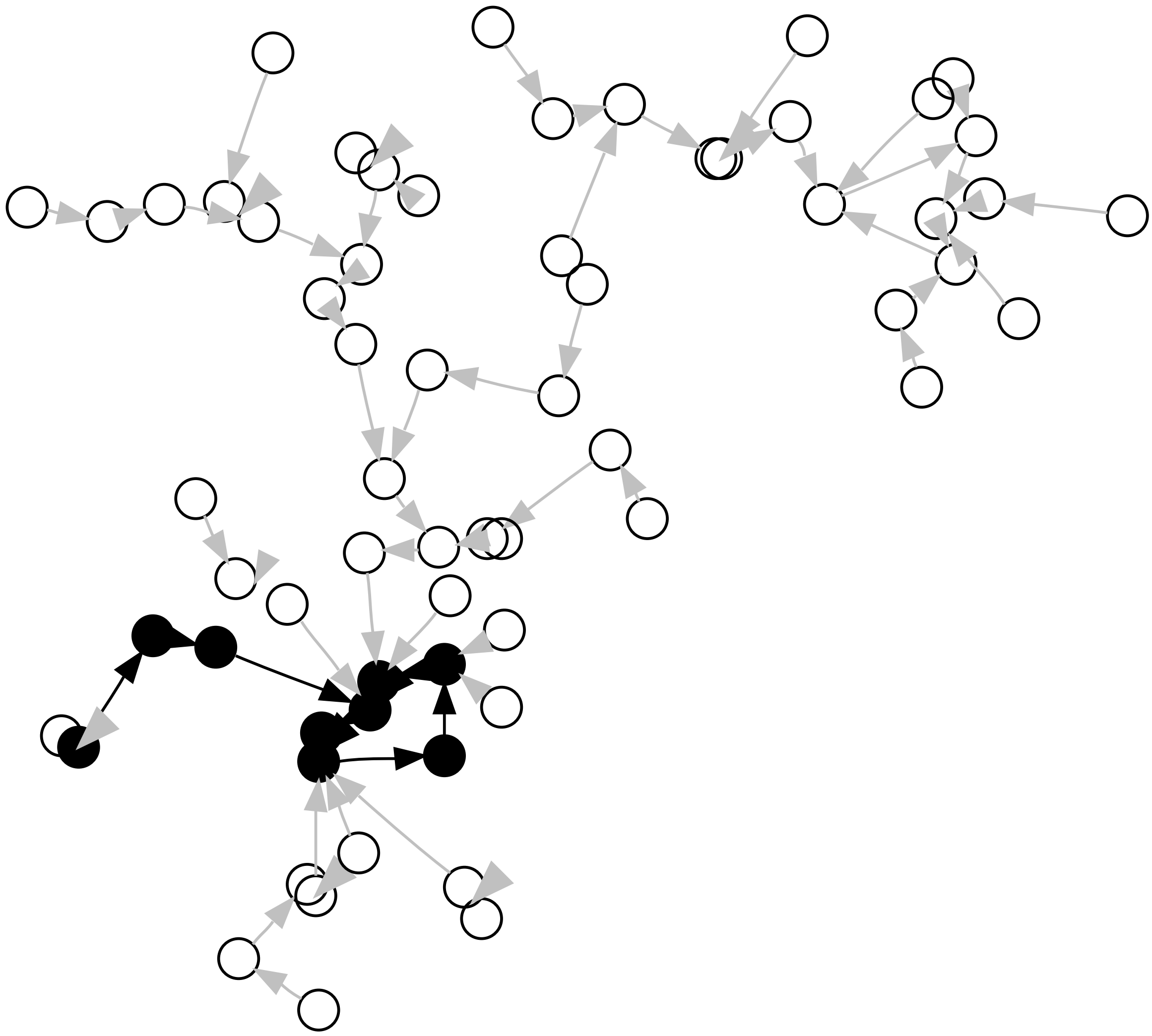


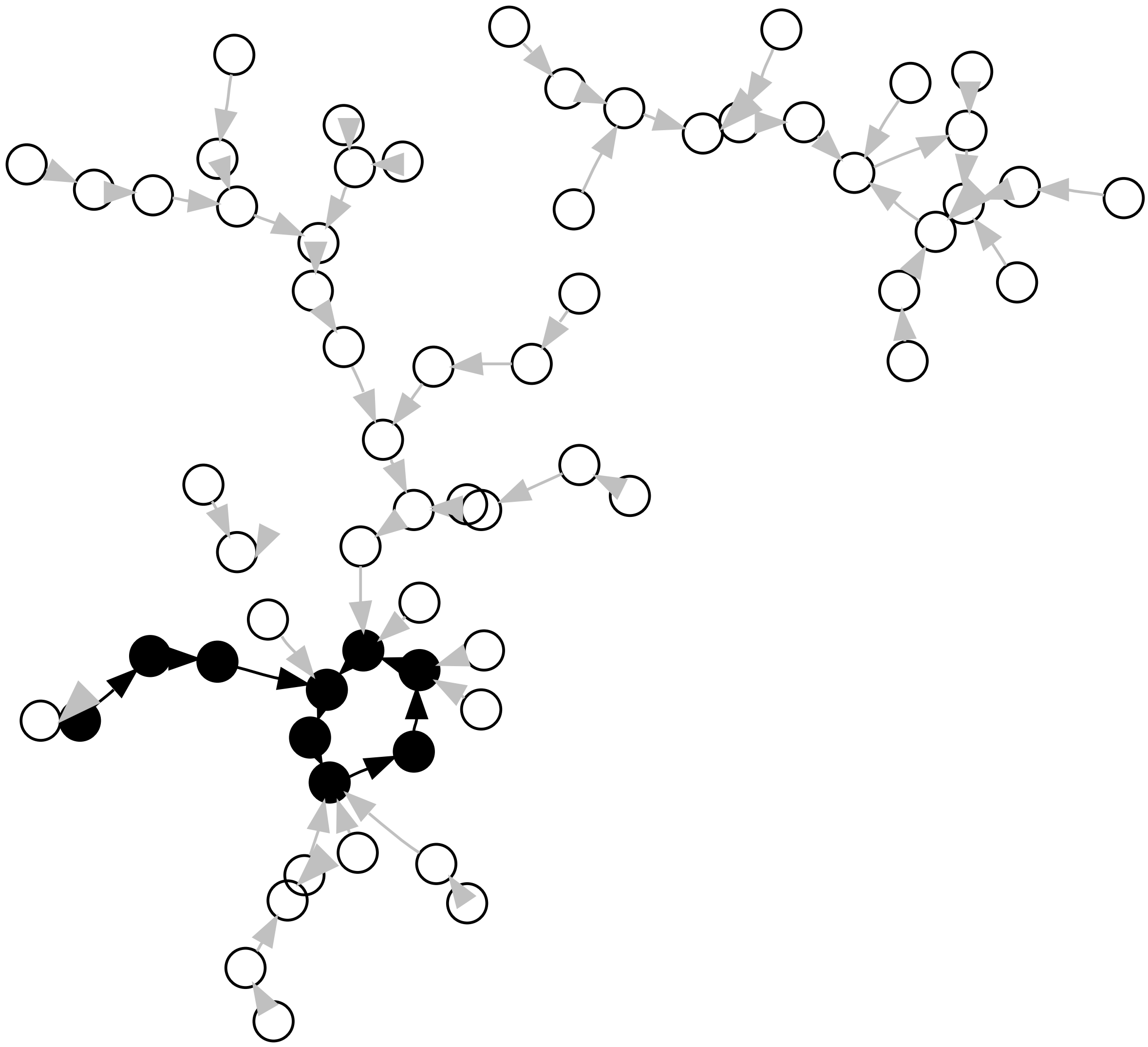


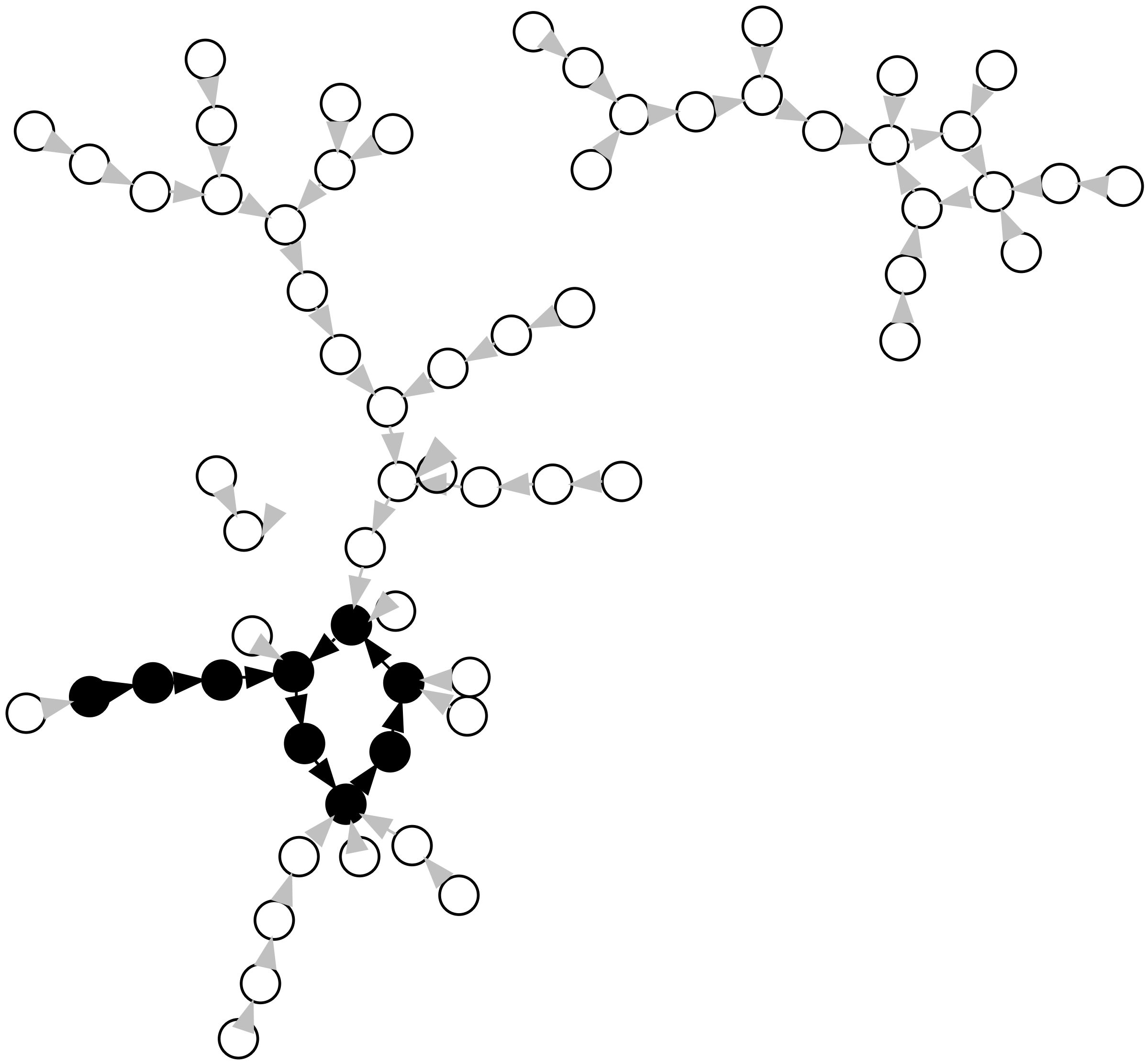












Assume that for each point
we know $a_i, b_i \in \mathbf{Z}/\ell\mathbf{Z}$
so that $W_i = [a_i]P + [b_i]Q$.

Then $W_i = W_j$ means that
$[a_i]P + [b_i]Q = [a_j]P + [b_j]Q$
so $[b_i - b_j]Q = [a_j - a_i]P$.
If $b_i \neq b_j$ the DLP is solved:
$k = (a_j - a_i)/(b_i - b_j)$.

Assume that for each point
we know $a_i, b_i \in \mathbf{Z}/\ell\mathbf{Z}$
so that $W_i = [a_i]P + [b_i]Q$.

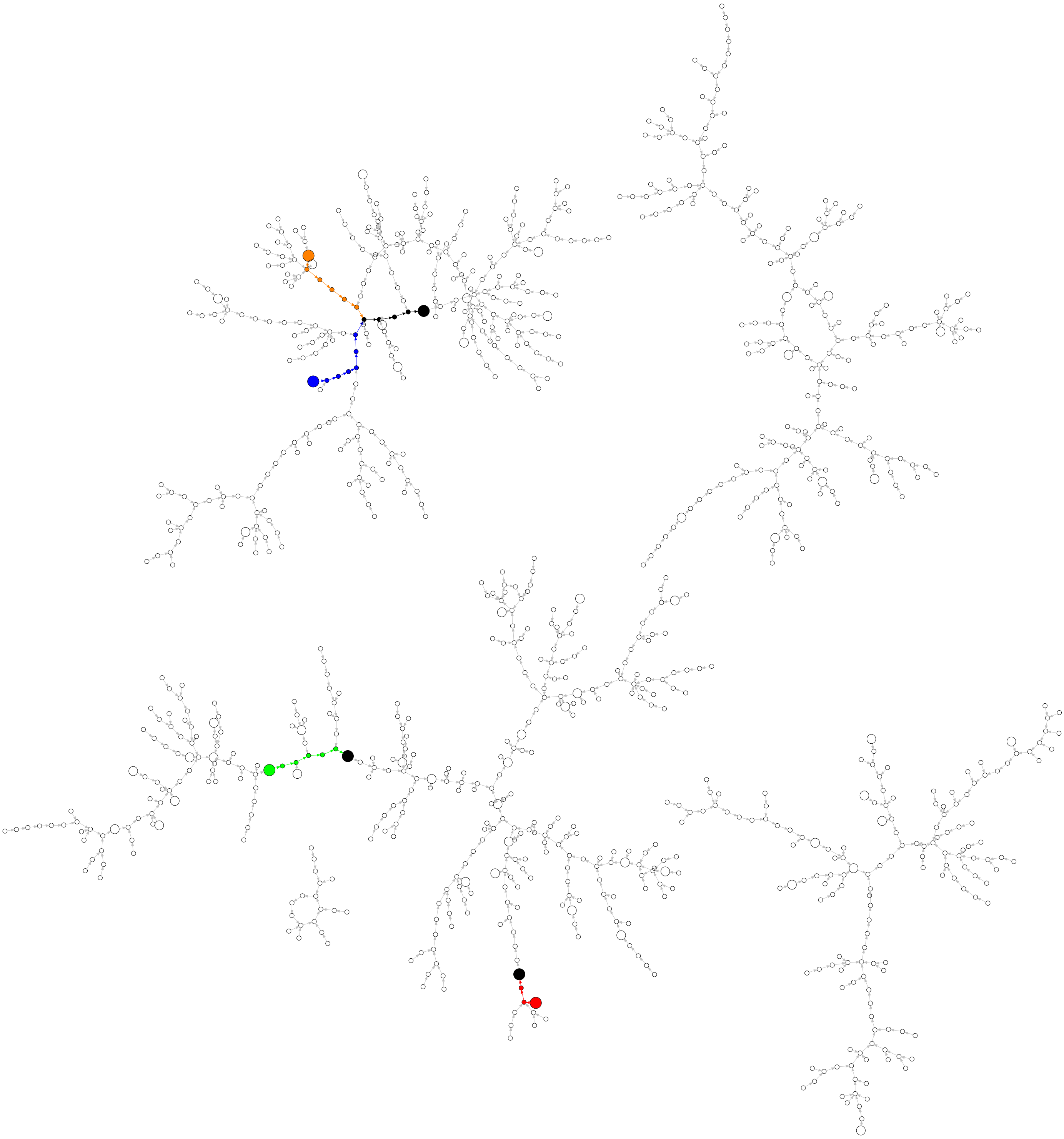
Then $W_i = W_j$ means that
$[a_i]P + [b_i]Q = [a_j]P + [b_j]Q$
so $[b_i - b_j]Q = [a_j - a_i]P$.
If $b_i \neq b_j$ the DLP is solved:
$k = (a_j - a_i)/(b_i - b_j)$.

e.g. "Additive walk":
Start with $W_0 = P$ and put
$f(W_i) = W_i + c_j P + d_j Q$
where $j = h(W_i)$.

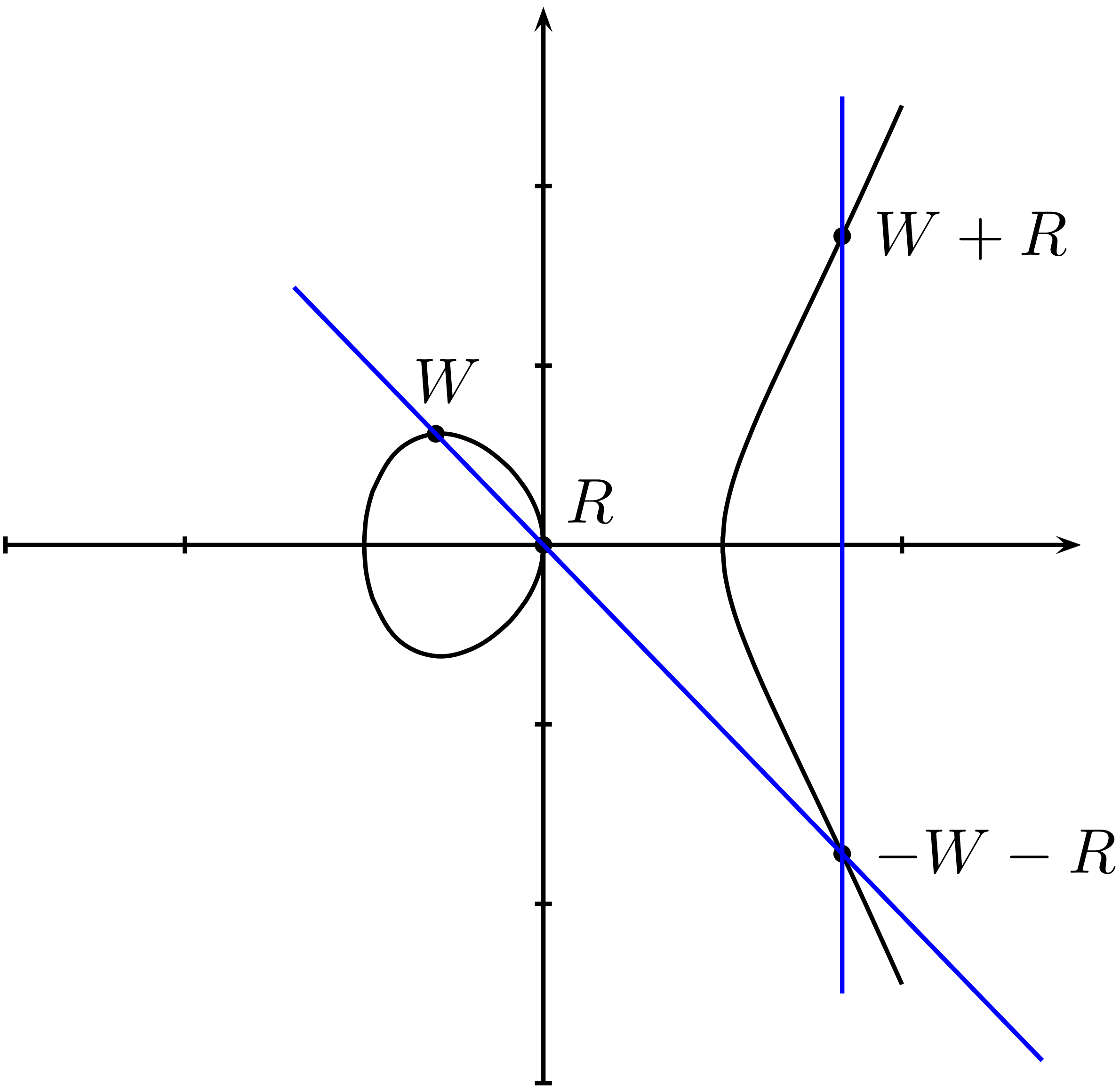
Parallel rho: Perform many walks with different starting points but same update function f.
If two different walks find the same point then their subsequent steps will match.

Terminate each walk once it hits a **distinguished point**.
Attacker chooses frequency and definition of distinguished points.
Do not wait for cycle.

Collect all distinguished points.
Two walks ending in same distinguished point solve DLP.

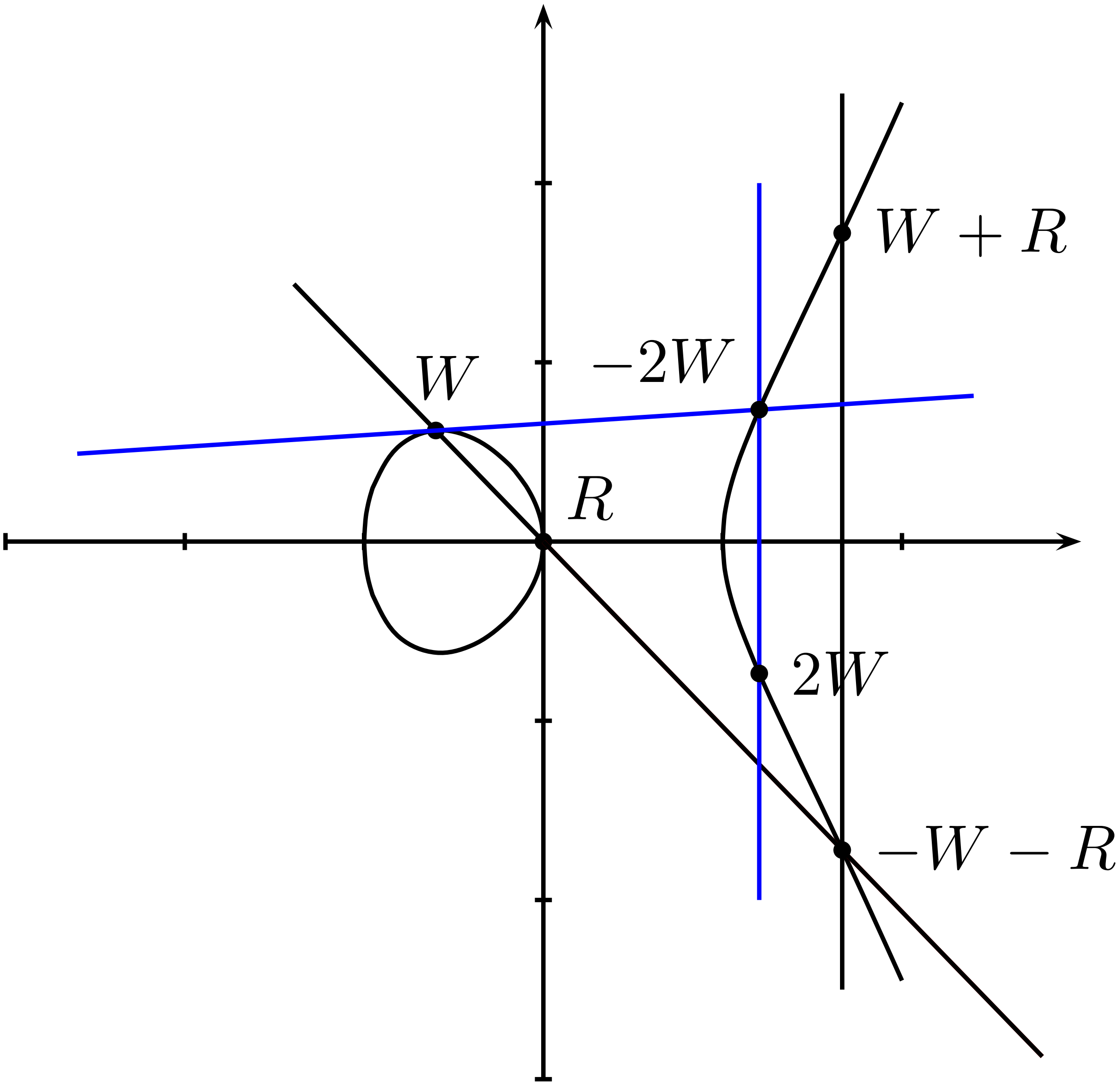


Elliptic-curve groups



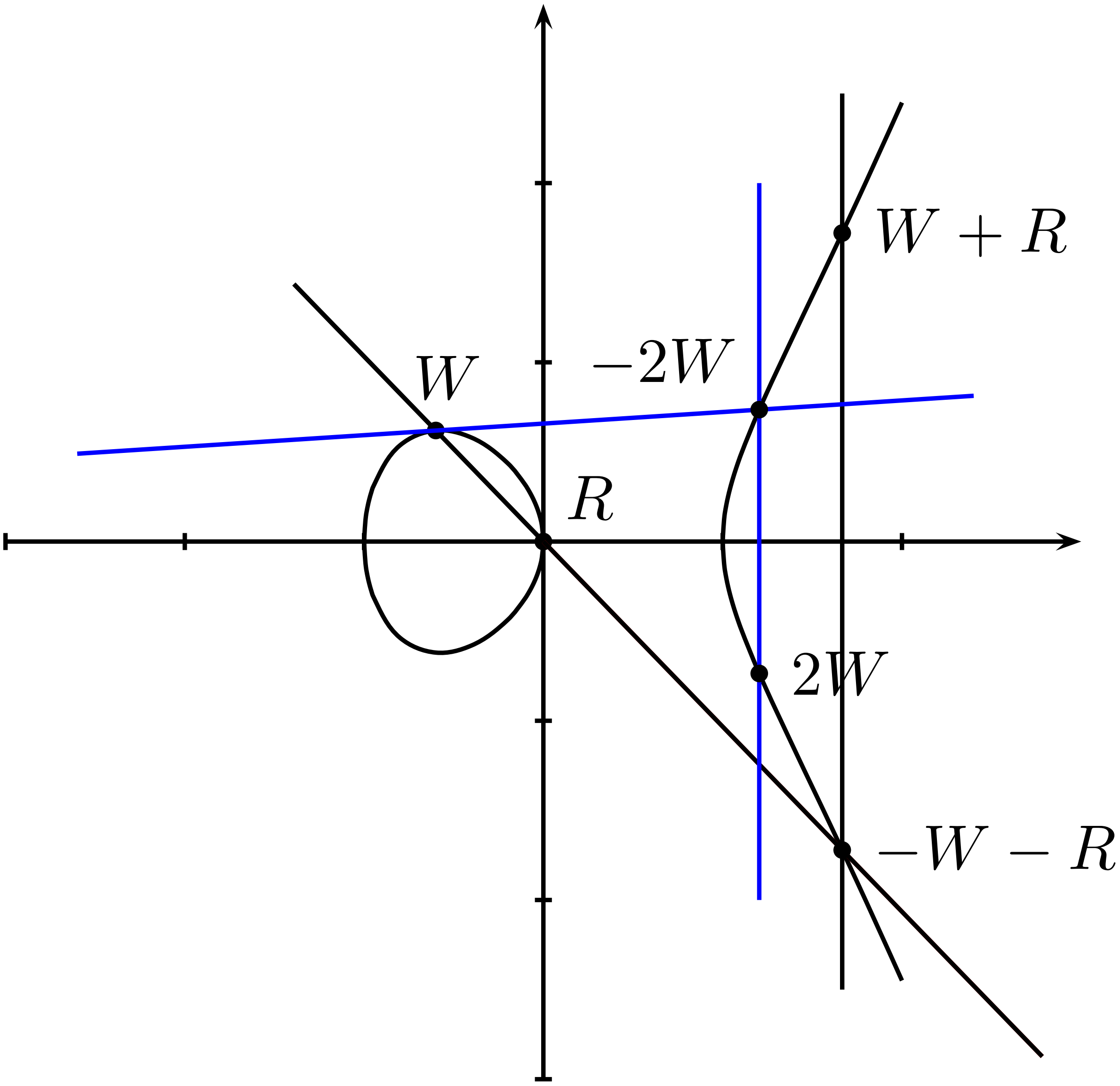
$$y^2 = x^3 + ax + b.$$

Elliptic-curve groups



$$y^2 = x^3 + ax + b.$$

Elliptic-curve groups



$y^2 = x^3 + ax + b$.

Also neutral element at ∞.

$-(x, y) = (x, -y)$.

$(x_W, y_W) + (x_R, y_R) =$
$(x_{W+R}, y_{W+R}) =$
$(\lambda^2 - x_W - x_R, \lambda(x_W - x_{W+R}) - y_W).$

$x_W \neq x_R$, "addition":
$\lambda = (y_R - y_W)/(x_R - x_W).$
Total cost $1\mathbf{I} + 2\mathbf{M} + 1\mathbf{S}$.

$W = R$ and $y_W \neq 0$, "doubling":
$\lambda = (3x_W^2 + a)/(2y_W).$
Total cost $1\mathbf{I} + 2\mathbf{M} + 2\mathbf{S}$.

Also handle some exceptions:
$(x_W, y_W) = (x_R, -y_R)$;
inputs at ∞.

For each prime $p \geq 3$
not dividing $4a^3 + 27b^2$:
Same formulas for $x, y \in \mathbf{F}_p$
define a group $E_{a,b}(\mathbf{F}_p)$.

Size of this group is element of
interval $[p+1-2\sqrt{p}, p+1+2\sqrt{p}]$.
"Random" element of interval
if a, b are random mod p.

Note 1: Some elliptic curves
do not have this form.

Note 2: For typical cryptographic
computations, much better
to use Edwards form instead.

Negation and rho

$W = (x, y)$ and $-W = (x, -y)$ have same x-coordinate. Search for x-coordinate collision.

Search space for collisions is only $\lceil \ell/2 \rceil$; this gives factor $\sqrt{2}$ speedup . . . if $f(W_i) = f(-W_i)$.

To ensure $f(W_i) = f(-W_i)$: Define $j = h(|W_i|)$ and $f(W_i) = |W_i| + c_j P + d_j Q$. Define $|W_i|$ as, e.g., lexicographic minimum of $W_i, -W_i$.

Problem: this walk can run into fruitless cycles!

Example: If $|W_{i+1}| = -W_{i+1}$ and $h(|W_{i+1}|) = j = h(|W_i|)$ then $W_{i+2} = f(W_{i+1}) = -W_{i+1} + c_j P + d_j Q = -(|W_i| + c_j P + d_j Q) + c_j P + d_j Q = -|W_i|$ so $|W_{i+2}| = |W_i|$
so $W_{i+3} = W_{i+1}$
so $W_{i+4} = W_{i+2}$ etc.

If h maps to r different values then expect this example to occur with probability $1/(2r)$ at each step.

Current ECDL record:

2009.07 Bos–Kaihara–
Kleinjung–Lenstra–Montgomery
"PlayStation 3 computing
breaks 2^{60} barrier:
112-bit prime ECDLP solved".

Standard curve over $\mathbf{F}_p$
where $p = (2^{128} - 3)/(11 \cdot 6949)$.

Current ECDL record:

2009.07 Bos–Kaihara–
Kleinjung–Lenstra–Montgomery
"PlayStation 3 computing
breaks 2^{60} barrier:
112-bit prime ECDLP solved".

Standard curve over $\mathbf{F}_p$
where $p = (2^{128} - 3)/(11 \cdot 6949)$.

"We did not use
the common negation map
since it requires branching
and results in code that runs
slower in a SIMD environment."
All modern CPUs are SIMD.

2009.07 Bos–Kaihara–Kleinjung–Lenstra–Montgomery "On the security of 1024-bit RSA and 160-bit elliptic curve cryptography":

Group order $q \approx p$; "expected number of iterations" is "$\sqrt{\frac{\pi \cdot q}{2}} \approx 8.4 \cdot 10^{16}$"; "we do not use the negation map"; "456 clock cycles per iteration per SPU"; "24-bit distinguishing property" $\Rightarrow$ "260 gigabytes".

"The overall calculation can be expected to take approximately 60 PS3 years."

2009.09 Bos–Kaihara–Montgomery "Pollard rho on the PlayStation 3":

"Our software implementation is optimized for the SPE . . . the computational overhead for [the negation map], due to the conditional branches required to check for fruitless cycles [13], results (in our implementation on this architecture) in an overall performance degradation."

"[13]" is 2000 Gallant–Lambert–Vanstone.

2010.07 Bos–Kleinjung–Lenstra "On the use of the negation map in the Pollard rho method":

"If the Pollard rho method is parallelized in SIMD fashion, it is a challenge to achieve any speedup at all. . . . Dealing with cycles entails administrative overhead and branching, which cause a non-negligible slowdown when running multiple walks in SIMD-parallel fashion. . . . [This] is a major obstacle to the negation map in SIMD environments."

Our software solves
random ECDL on the same curve
(with no precomputation)
in 35.6 PS3 years on average.

For comparison:
Bos–Kaihara–Kleinjung–Lenstra–
Montgomery software
uses 65 PS3 years on average.

Our software solves
random ECDL on the same curve
(with no precomputation)
in 35.6 PS3 years on average.

For comparison:
Bos–Kaihara–Kleinjung–Lenstra–
Montgomery software
uses 65 PS3 years on average.

Computation used 158000 kWh
(if PS3 ran at only 300W),
wasting $>$70000 kWh,
unnecessarily generating $>$10000
kilograms of carbon dioxide.
(0.143 kg CO2 per Swiss kWh.)

Several levels of speedups,
starting with fast arithmetic
mod $p = (2^{128} - 3)/(11 \cdot 6949)$
and continuing up through rho.

Most important speedup:
We use the negation map.

Several levels of speedups,
starting with fast arithmetic
mod $p = (2^{128} - 3)/(11 \cdot 6949)$
and continuing up through rho.

Most important speedup:
We use the negation map.

Extra cost in each iteration:
extract bit of "s"
(normalized y, needed anyway);
expand bit into mask;
use mask to conditionally
replace (s, y) by $(-s, -y)$.
5.5 SPU cycles ($\approx 1.5\%$ of total).
No conditional branches.

Bos–Kleinjung–Lenstra say that “on average more elliptic curve group operations are required per step of each walk. This is unavoidable” etc.

Specifically: If the precomputed additive-walk table has r points, need 1 extra doubling to escape a cycle after $\approx 2r$ additions. And more: “cycle reduction” etc.

Bos–Kleinjung–Lenstra say that the benefit of large r is “wiped out by cache inefficiencies.”

There's really no problem here!

We use $r = 2048$.
$1/(2r) = 1/4096$; negligible.

Recall: p has 112 bits.
28 bytes for table entry (x, y).

We expand to 36 bytes
to accelerate arithmetic.
We compress to 32 bytes
by insisting on small x, y;
very fast initial computation.

Only 64KB for table.
Our Cell table-load cost: 0,
overlapping loads with arithmetic.
No "cache inefficiencies."

What about fruitless cycles?

We run 45 iterations.
We then save s;
run 2 slightly slower iterations
tracking minimum (s, x, y);
then double tracked (x, y)
if new s equals saved s.

(Occasionally replace 2 by 12
to detect 4-cycles, 6-cycles.
Such cycles are almost
too rare to worry about,
but detecting them has a
completely negligible cost.)

Maybe fruitless cycles waste
some of the 47 iterations.
. . . but this is infrequent.
Lose $\approx$ 0.6% of all iterations.

Tracking minimum isn't free,
but most iterations skip it!
Same for final s comparison.
Still no conditional branches.
Overall cost $\approx$ 1.3%.

Doubling occurs for only
$\approx$ 1/4096 of all iterations.
We use SIMD quite lazily here;
overall cost $\approx$ 0.6%.
Can reduce this cost further.

Are we sure about all this?
Are there hidden bottlenecks?
Are we accidentally
compromising walk randomness?

Are we sure about all this?
Are there hidden bottlenecks?
Are we accidentally
compromising walk randomness?

Check by running experiments!
e.g. Try 1000 experiments;
check that average time
is very close to our predictions.

Are we sure about all this?
Are there hidden bottlenecks?
Are we accidentally
compromising walk randomness?

Check by running experiments!
e.g. Try 1000 experiments;
check that average time
is very close to our predictions.

Problem: 1000 experiments
should take 35600 PS3 years.
We don't have many PS3s.

Are we sure about all this?
Are there hidden bottlenecks?
Are we accidentally
compromising walk randomness?

Check by running experiments!
e.g. Try 1000 experiments;
check that average time
is very close to our predictions.

Problem: 1000 experiments
should take 35600 PS3 years.
We don't have many PS3s.

Solution: Try same algorithm
at some smaller scales.

Our software works for
any curve $y^2 = x^3 - 3x + b$
over the same $\mathbf{F}_p$.
Same cost of field arithmetic,
same cost of curve arithmetic.

$y^2 = x^3 - 3x + 238^2$
has a point of order $\approx 2^{50}$.

$y^2 = x^3 - 3x + 372^2$
has a point of order $\approx 2^{55}$.

$y^2 = x^3 - 3x + 240^2$
has a point of order $\approx 2^{60}$.

We tried > 32000 experiments
on each of these curves.

Found distinguished points
at the predicted rates.

Found discrete logarithms
using the predicted number
of distinguished points.

Negation conclusions:
Sensible use of negation,
with or without SIMD,
has negligible impact
on cost of each iteration.
Impact on number of iterations
is almost exactly $\sqrt{2}$.
Overall benefit is
extremely close to $\sqrt{2}$.

How to evaluate security for sparse families?

Get people to solve big challenges!

1997: Certicom announces several elliptic-curve challenges.

"The Challenge is to compute the ECC private keys from the given list of ECC public keys
and associated system parameters.
This is the type of problem
facing an adversary who
wishes to completely defeat
an elliptic curve cryptosystem."

Goals: help users select key sizes; compare random and Koblitz; compare $\mathbf{F}_{2^m}$ and $\mathbf{F}_p$; etc.

How to get them hooked?

1997: ECCp-79 broken by
Baisley and Harley.

1997: ECC2-79 broken by
Harley et al.

1998: ECCp-89, ECC2-89 broken
by Harley et al.

1998: ECCp-97 broken by Harley
et al. (1288 computers).

1998: ECC2K-95 broken by Harley
et al. (200 computers).

1999: ECC2-97 broken by Harley
et al. (740 computers).

2000: ECC2K-108 broken by Harley
et al. (9500 computers).

More challenging challenges

Certicom: "The 109-bit Level I challenges are feasible using a very large network of computers. The 131-bit Level I challenges are expected to be infeasible against realistic software and hardware attacks, unless of course, a new algorithm for the ECDLP is discovered."

2002: ECCp-109 broken by Monico et al. (10000 computers).

2004: ECC2-109 broken by Monico et al. (2600 computers).

open: ECC2K-130

With our latest implementations,
ECC2K-130 is breakable
in two years on average by

- 1595 Phenom II x4 955 CPUs,

With our latest implementations,
ECC2K-130 is breakable
in two years on average by

- 1595 Phenom II x4 955 CPUs,
- or 1231 Playstation 3s,

With our latest implementations,
ECC2K-130 is breakable
in two years on average by

- 1595 Phenom II x4 955 CPUs,
- or 1231 Playstation 3s,
- or 534 GTX 295 cards,

With our latest implementations,
ECC2K-130 is breakable
in two years on average by

- 1595 Phenom II x4 955 CPUs,
- or 1231 Playstation 3s,
- or 534 GTX 295 cards,
- or 308 XC3S5000 FPGAs,

With our latest implementations,
ECC2K-130 is breakable
in two years on average by

- 1595 Phenom II x4 955 CPUs,
- or 1231 Playstation 3s,
- or 534 GTX 295 cards,
- or 308 XC3S5000 FPGAs,
- or any combination thereof.

With our latest implementations,
ECC2K-130 is breakable
in two years on average by

- 1595 Phenom II x4 955 CPUs,
- or 1231 Playstation 3s,
- or 534 GTX 295 cards,
- or 308 XC3S5000 FPGAs,
- or any combination thereof.

This is a computation that
Certicom called "infeasible"?

With our latest implementations, ECC2K-130 is breakable in two years on average by

- 1595 Phenom II x4 955 CPUs,
- or 1231 Playstation 3s,
- or 534 GTX 295 cards,
- or 308 XC3S5000 FPGAs,
- or any combination thereof.

This is a computation that Certicom called “infeasible”?

Certicom has now backpedaled, saying that ECC2K-130 “may be within reach”.

The target: ECC2K-130

The Koblitz curve
$y^2 + xy = x^3 + 1$
over $\mathbf{F}_{2^{131}}$ has 4ℓ points,
where ℓ is prime.
Field representation uses
irreducible polynomial
$f = z^{131} + z^{13} + z^2 + z + 1$.

Certicom generated their
challenge points as two random
points in order-ℓ subgroup by
taking two random points on the
curve and multiplying them by 4.

This produced the following points P, Q:

```
x(P) = 05 1C99BFA6 F18DE467 C80C23B9 8C7994AA
y(P) = 04 2EA2D112 ECEC71FC F7E000D7 EFC978BD
x(Q) = 06 C997F3E7 F2C66A4A 5D2FDA13 756A37B1
y(Q) = 04 A38D1182 9D32D347 BD0C0F58 4D546E9A
```

(unique encoding of $\mathbf{F}_{2^{131}}$ in hex).

The challenge:
Find an integer
$k \in \{0, 1, \ldots, \ell - 1\}$
such that $[k]P = Q$.

Bigger picture:
128-bit curves have been proposed for real (RFID, TinyTate).

Equivalence classes for Koblitz curves

P and $-P$ have same x-coordinate.
Search for x-coordinate collision.
Search space is only $\ell/2$; this gives factor $\sqrt{2}$ speedup . . .
provided that $f(P_i) = f(-P_i)$.

Equivalence classes for Koblitz curves

P and $-P$ have same x-coordinate.
Search for x-coordinate collision.
Search space is only $\ell/2$; this gives factor $\sqrt{2}$ speedup ... provided that $f(P_i) = f(-P_i)$.

More savings: P and $\sigma^i(P)$ have $x(\sigma^j(P)) = x(P)^{2^j}$.

Consider equivalence classes under Frobenius and $\pm$;
gain factor $\sqrt{2n} = \sqrt{2 \cdot 131}$.

Need to ensure that the iteration function satisfies
$f(P_i) = f(\pm\sigma^j(P_i))$ for any j.

Savings is $\sqrt{2 \cdot 131}$ iterations—
but the iteration function
has become slower.
How much slower?

Savings is $\sqrt{2 \cdot 131}$ iterations—
but the iteration function
has become slower.
How much slower?

Could again define adding walk
starting from $|P_i|$.

Redefine $|P_i|$ as canonical
representative of class containing
P_i: e.g., lexicographic minimum
of P_i, $-P_i$, $\sigma(P_i)$, etc.

Iterations now involve many
squarings, but squarings are not
so expensive in characteristic 2.

Iteration function for Koblitz curves

Normal basis of finite field $\mathbf{F}_{2^n}$ has elements $\{\zeta, \zeta^2, \zeta^{2^2}, \zeta^{2^3}, \ldots, \zeta^{2^{n-1}}\}$.

Representation for x and x^2

$\sum_{i=0}^{n-1} x_i \zeta^{2^i} = (x_0, x_1, x_2, \ldots, x_{n-1})$

$\sum_{i=1}^{n} x_i \zeta^{2^i} = (x_{n-1}, x_0, \ldots, x_{n-2})$

using $(\zeta^{2^{n-1}})^2 = \zeta^{2^n} = \zeta$.

Harley and Gallant-Lambert-Vanstone use that in normal basis, $x(P)$ and $x(P)^{2^j}$ have same Hamming weight

$$\text{HW}(x(P)) = \sum_{i=0}^{n-1} x_i$$

(addition over $\mathbf{Z}$).

Suggestion:

$$P_{i+1} = P_i + \sigma^j(P_i),$$

as iteration function.

Choice of j depends on HW($x(P)$).

This ensures that the walk is well defined on classes since

$$f(\pm\sigma^m(P_i)) =$$
$$\pm\,\sigma^m(P_i) + \sigma^j(\pm\sigma^m(P_i)) =$$
$$\pm\,(\sigma^m(P_i) + \sigma^m(\sigma^j(P_i))) =$$
$$\pm\,\sigma^m(P_i + \sigma^j(P_i)) =$$
$$\pm\,\sigma^m(P_{i+1}).$$

GLV suggest using
$j = \text{hash}(\text{HW}(x(P)))$,
where the hash function
maps to $[1, n]$.

Harley uses a smaller set of
exponents; for his attack on
ECC2K-108 he takes
$j \in \{1, 2, 4, 5, 6, 7, 8\}$;
computed as
$j = (\text{HW}(x(P)) \bmod 7) + 2$
and replacing 3 by 1.

Our choice of iteration function

Restricting size of j matters—
squarings are cheap but:

- in bitslicing need to compute all powers (no branches allowed);
- code size matters
  (in particular for Cell CPU);
- logic costs area for FPGA;
- having a large set doesn't actually gain much randomness.

Optimization target:
time per iteration $\times$ # iterations.

How to mention lattices?

Having few coefficients lets us exclude short fruitless cycles.

To do so, compute
the shortest vector in the lattice

$$\left\{v : \prod_j (1+\sigma^j)^{v_j} = 1\right\}.$$

Usually the shortest vector has negative coefficients (which cannot happen with the iteration); shortest vector with positive coefficients is somewhat longer.

For implementation it is better to have a continuous interval of exponents, so shift the interval if shortest vector is short.

Our iteration function:

$P_{i+1} = P_i + \sigma^j(P_i)$ where

$j = (\mathrm{HW}(x(P))/2 \bmod 8) + 3$,

so $j \in \{3, 4, 5, 6, 7, 8, 9, 10\}$.

Shortest combination of these powers is long.

Note that $\mathrm{HW}(x(P))$ is even.

Iteration consists of

- computing the Hamming weight $\mathrm{HW}(x(P))$ of the normal-basis representation of $x(P)$;
- checking for distinguished points (is $\mathrm{HW}(x(P)) \leq 34$?);
- computing j and $P + \sigma^j(P)$.

Analysis of our iteration function

For a perfectly random walk
$\approx \sqrt{\pi \ell / 2}$ iterations
are expected on average.
Have $\ell \approx 2^{131}/4$ for ECC2K-130.

A perfectly random walk
on classes under $\pm$ and Frobenius
would reduce number of iterations
by $\sqrt{2 \cdot 131}$.

Analysis of our iteration function

For a perfectly random walk
$\approx \sqrt{\pi \ell / 2}$ iterations
are expected on average.
Have $\ell \approx 2^{131}/4$ for ECC2K-130.

A perfectly random walk
on classes under $\pm$ and Frobenius
would reduce number of iterations
by $\sqrt{2 \cdot 131}$.

Loss of randomness
from having only 8 choices of j.
Further loss from non-randomness
of Hamming weights:

Hamming weights around 66
are much more likely than at the
edges; effect still noticeable
after reduction to 8 choices.

Hamming weights around 66
are much more likely than at the
edges; effect still noticeable
after reduction to 8 choices.

Our $\sqrt{1 - \sum_i p_i^2}$ heuristic says
that the total loss is 6.9993%.
(Higher-order anti-collision
analysis: actually above 7%.)
This loss is justified by
the very fast iteration function.

Hamming weights around 66
are much more likely than at the
edges; effect still noticeable
after reduction to 8 choices.

Our $\sqrt{1-\sum_i p_i^2}$ heuristic says
that the total loss is 6.9993%.
(Higher-order anti-collision
analysis: actually above 7%.)
This loss is justified by
the very fast iteration function.

Average number of iterations for
our attack against ECC2K-130:
$\sqrt{\pi\ell/(2\cdot 2\cdot 131)}\cdot 1.069993$
$\approx 2^{60.9}$.

Endomorphisms

In general, an efficiently computable endomorphism ϕ of order r speeds up Pollard rho method by factor $\sqrt{r}$.
This theoretical speedup can usually be realized in practice—it just requires some work.

Can define walk on classes by inspecting all $2r$ points
$\pm P, \pm\phi(P), \ldots, \pm\phi^{r-1}(P)$
to choose unique representative for class and then doing an adding walk; but this is slow.

What is the security of ECC2K-130?

How long do $\approx 2^{60.9}$ iterations take?

What is the security of ECC2K-130?

How long do $\approx 2^{60.9}$ iterations take? $70110 \cdot 2^{60.9}$ bit operations!

What is the security of ECC2K-130?

How long do $\approx 2^{60.9}$ iterations take? $70110 \cdot 2^{60.9}$ bit operations!

Time?

What is the security of ECC2K-130?

How long do $\approx 2^{60.9}$ iterations take? $70110 \cdot 2^{60.9}$ bit operations! Time? Depends on platform; hardware has area-time tradeoffs; software does not work on bits!

What is the security of ECC2K-130?

How long do $\approx 2^{60.9}$ iterations take? $70110 \cdot 2^{60.9}$ bit operations! Time? Depends on platform; hardware has area-time tradeoffs; software does not work on bits!

Need implementations on different platforms with low-level optimizations.

What is the security of ECC2K-130?

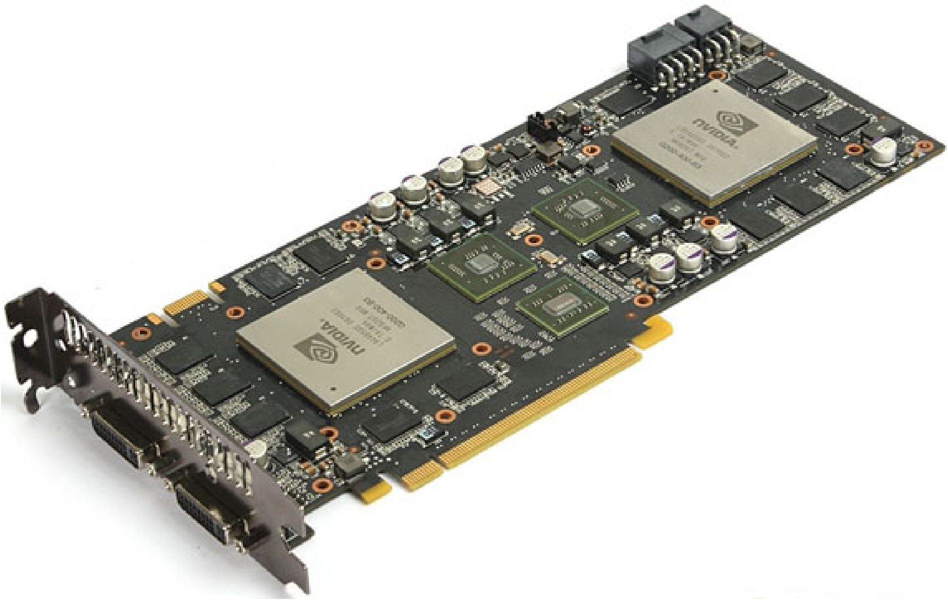
How long do $\approx 2^{60.9}$ iterations take? $70110 \cdot 2^{60.9}$ bit operations! Time? Depends on platform; hardware has area-time tradeoffs; software does not work on bits!

Need implementations on different platforms with low-level optimizations.

bit operations gives good indication for complexity on FPGAs; is also meaningful for speed of bitsliced software.

Graphics cards

GTX 295 without fans, case:



Overclocked Radeon 5970:

Why GPUs are interesting

NVIDIA GTX 295 graphics card
has two GPUs.
Each GPU has 30 cores
running at 1.242GHz.
(NVIDIA: "30 multiprocessors.")

Each core can perform
8 32-bit operations/cycle.
Total GTX 295 power:
480 32-bit ops/cycle.
(NVIDIA: "480 cores.")

$> 2^{39}$ 32-bit ops/second.
$> 2^{69}$ 1-bit ops/year.

Compare to Cell SPEs:

6 cores running at 3.2GHz.
Each core can perform
4 32-bit operations/cycle.
Total power:
24 32-bit ops/cycle.

Despite low clock speed,
GTX 295 can do $> 7\times$ more
operations/second than Cell.
Similar price to Cell.

Newer GPUs are even faster.

Why GPUs are difficult

GPU core issues each instruction to many threads. Using full GPU power is difficult with < 192 threads, impossible with < 128 threads.

All data used by these threads must fit into core's SRAM: 65536 bytes of registers, 16384 bytes of shared memory.

Copying data from DRAM has huge latency, low throughput.

GPU results

Best speed with NVIDIA compiler:
$\approx$ 3000 cycles/iteration.

Gave up on compiler, built
new GPU assembly language,
rewrote the software:
1379 cycles/iteration.

Current software:
1164 cycles/iteration.

GPU results

Best speed with NVIDIA compiler:
$\approx$ 3000 cycles/iteration.

Gave up on compiler, built
new GPU assembly language,
rewrote the software:
1379 cycles/iteration.

Current software:
1164 cycles/iteration.

Lower bound for arithmetic:
273 cycles/iteration.
Main slowdown: loads + stores.

Need 534 GPUs for 2 years.

Need 534 GPUs for 2 years.

World of Warcraft:
10 million subscribers
who invest heavily in
their own graphics cards.

Need 534 GPUs for 2 years.

World of Warcraft:
10 million subscribers
who invest heavily in
their own graphics cards.

$534 \cdot 2 \cdot 365 \cdot 24$
$= 9\,355\,680 < 10\,000\,000.$

Need 534 GPUs for 2 years.

World of Warcraft:
10 million subscribers
who invest heavily in
their own graphics cards.

$534 \cdot 2 \cdot 365 \cdot 24$
$= 9\,355\,680 < 10\,000\,000.$

All we need is
1 hour of World of Warcraft!