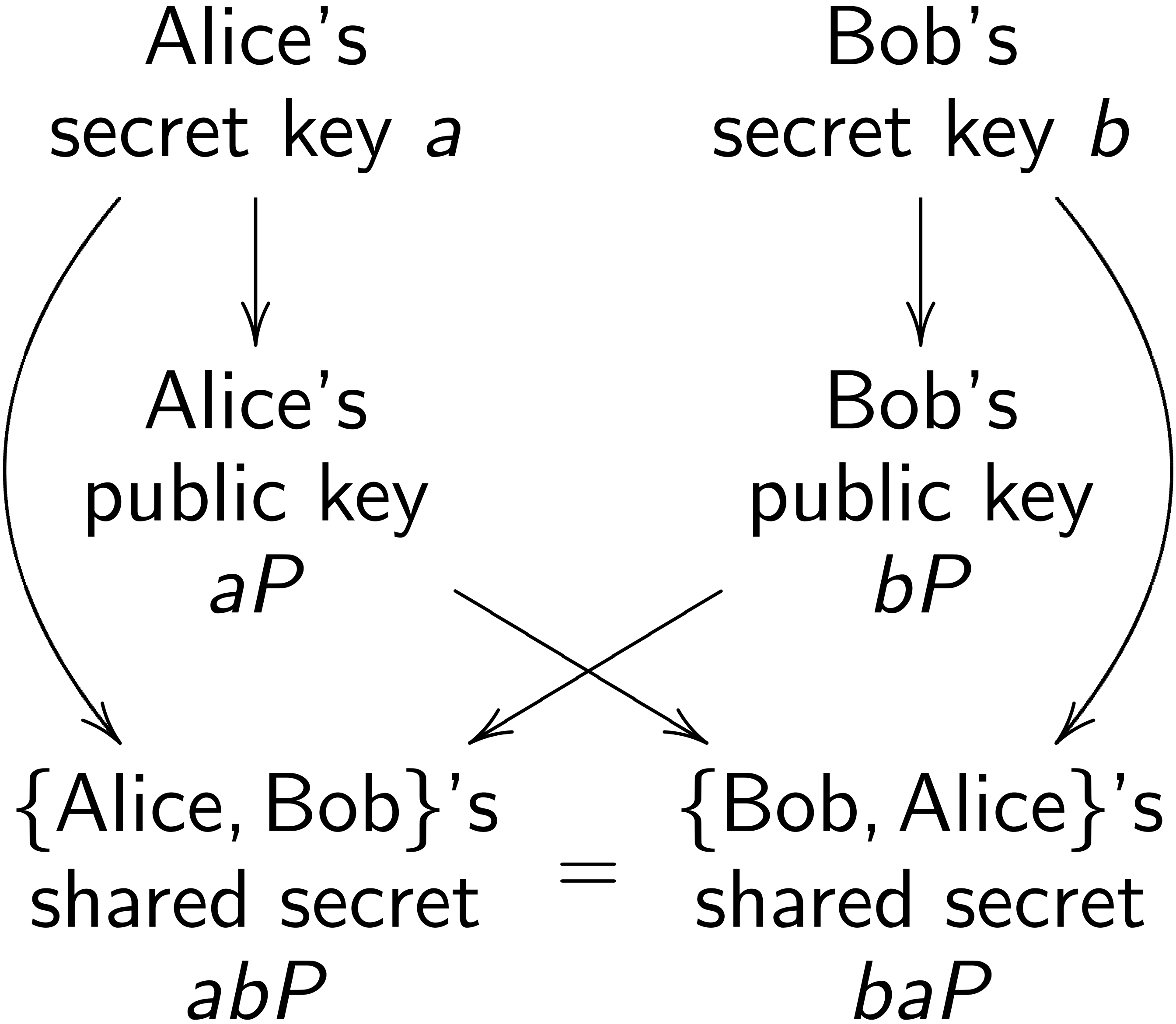
How to manipulate curve standards: a white paper for the black hat

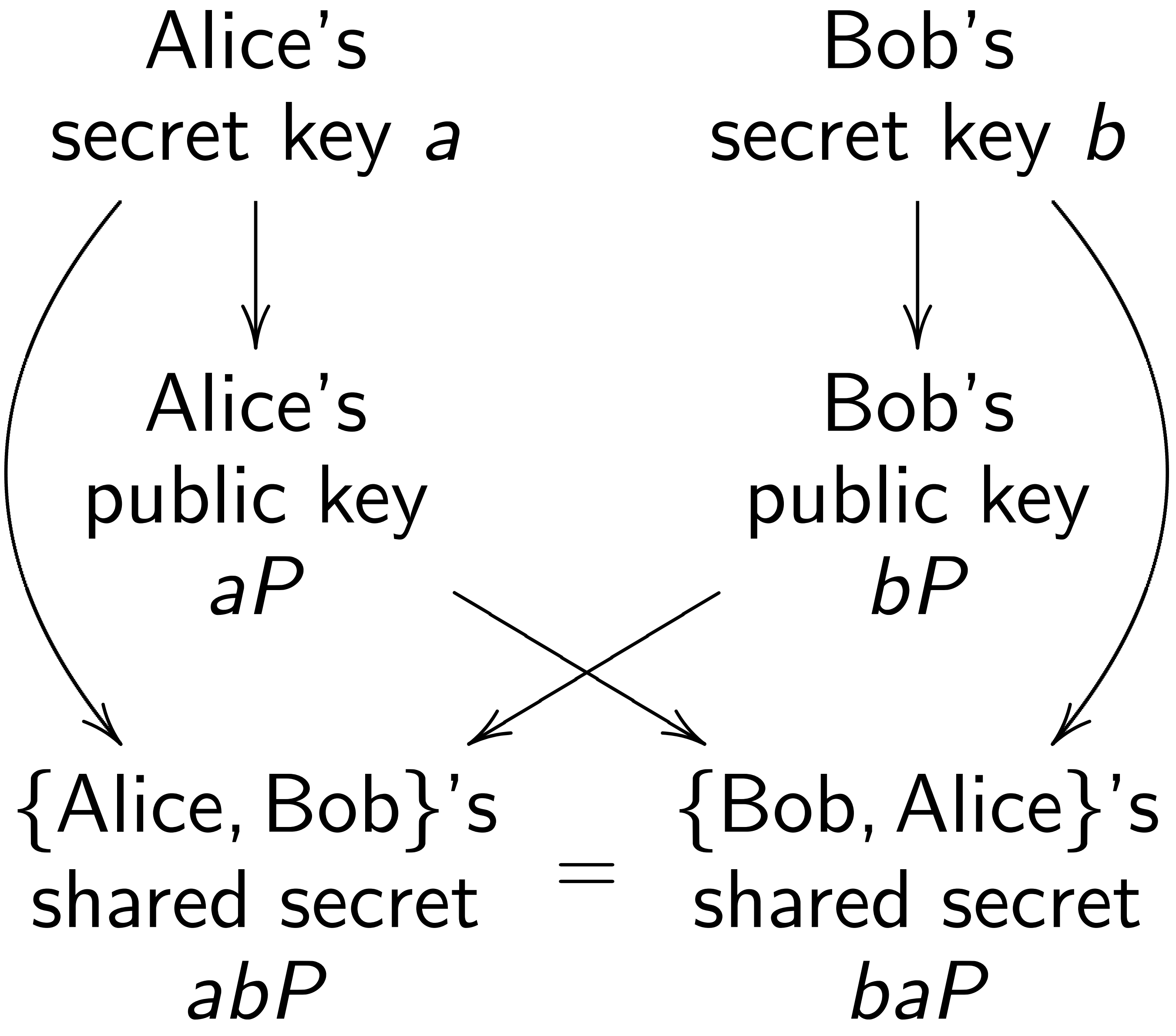
Daniel J. Bernstein
Tung Chou
Chitchanok Chuengsatiansup
Andreas Hülsing
Eran Lambooij
Tanja Lange
Ruben Niederhagen
Christine van Vredendaal

bada55.cr.yp.to

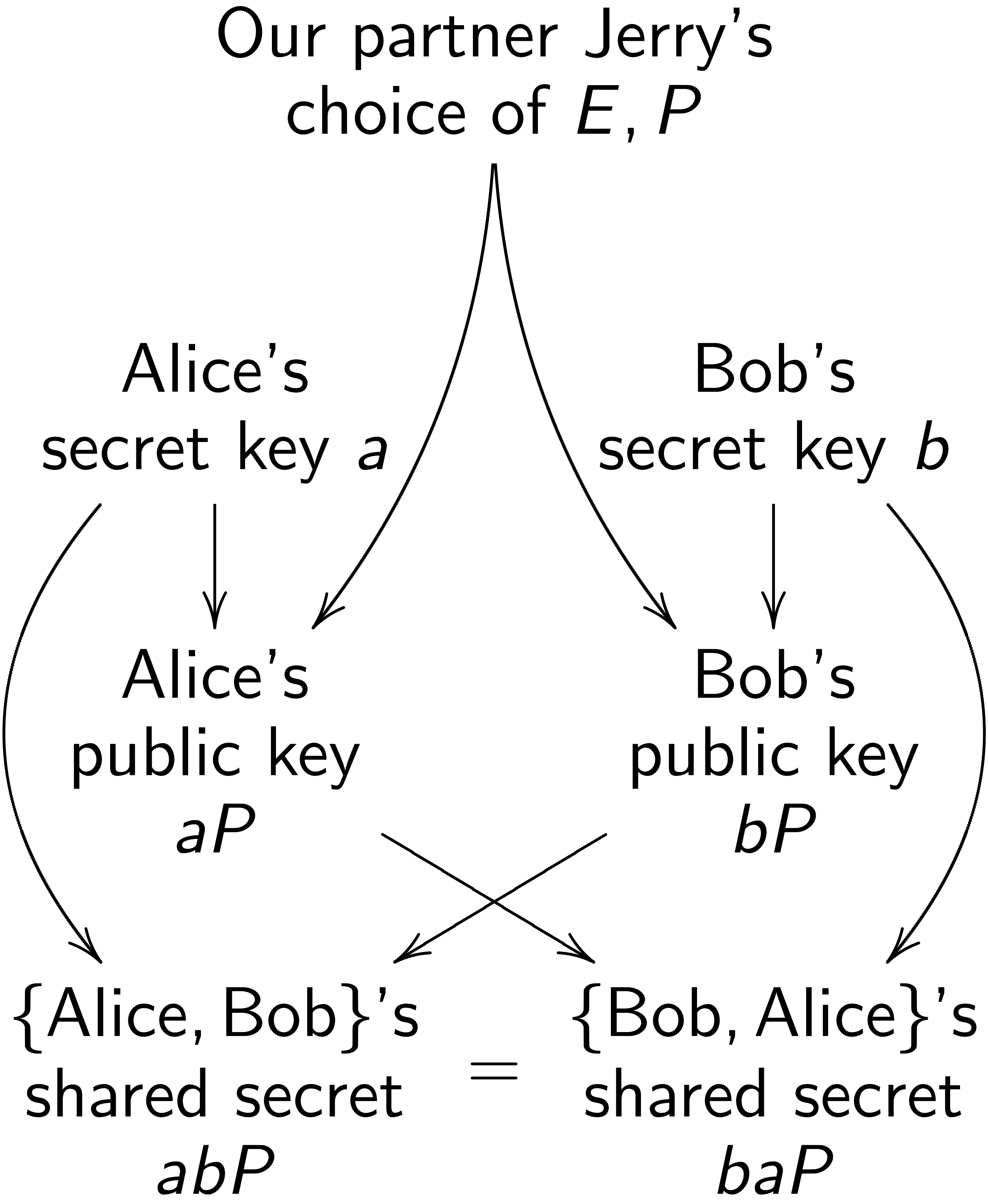
Textbook key exchange
using standard point P
on a standard elliptic curve E:

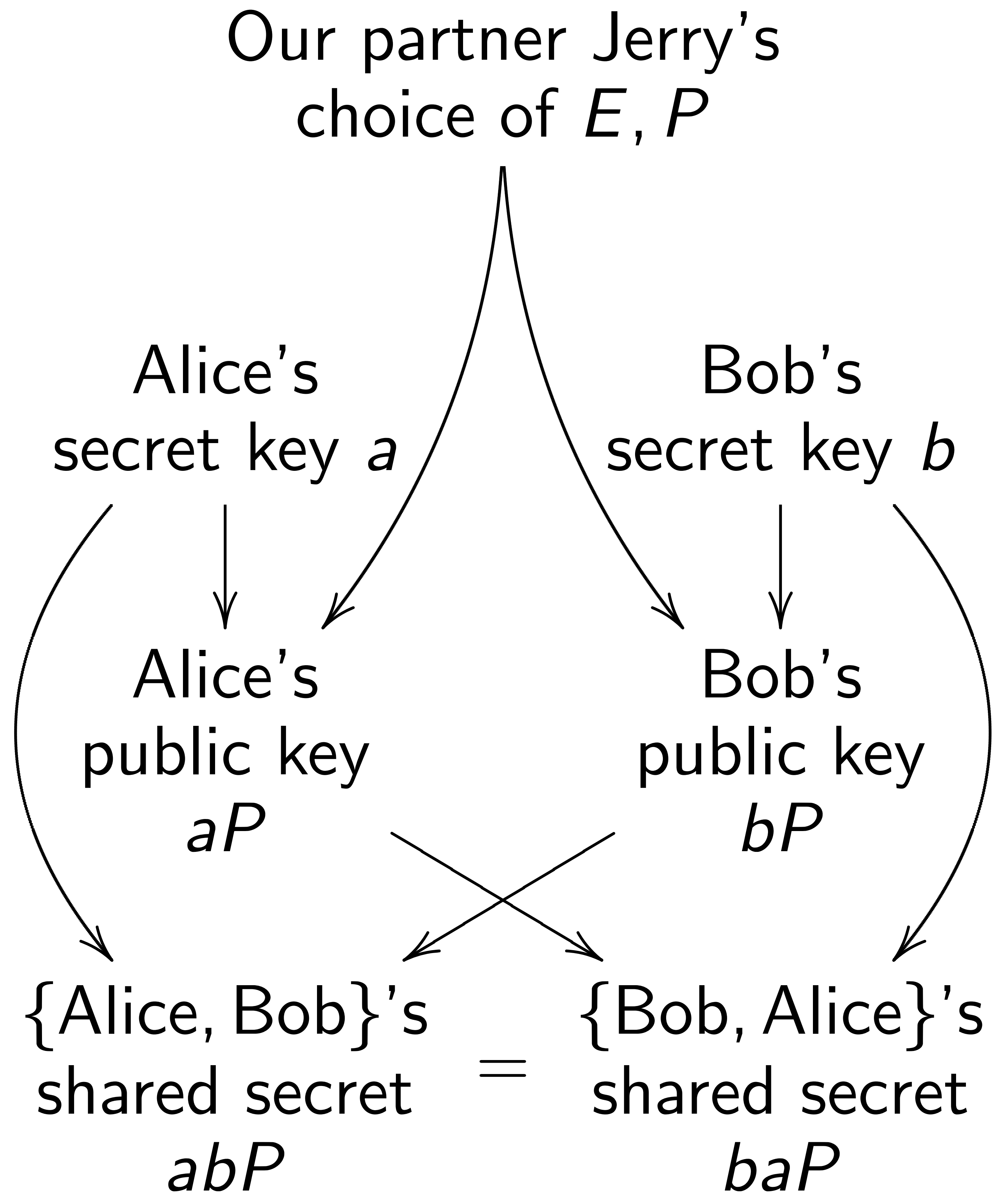


Textbook key exchange
using standard point P
on a standard elliptic curve E:



Security depends on choice of E.


Our partner Jerry's choice of E, P
Alice's secret key a
Bob's secret key b
Alice's public key aP
Bob's public key bP
$\{$Alice, Bob$\}$'s shared secret abP
=
$\{$Bob, Alice$\}$'s shared secret baP



Can we exploit this picture?

Exploitability depends on public criteria for accepting E, P.

Exploitability depends on
public criteria for accepting E, P.

Extensive ECC literature:
Pollard rho breaks small E,
Pohlig–Hellman breaks most E,
MOV/FR breaks some E,
SmartASS breaks some E, etc.

Assume that public will accept
any E not publicly broken.

Exploitability depends on public criteria for accepting E, P.

Extensive ECC literature:
Pollard rho breaks small E,
Pohlig–Hellman breaks most E,
MOV/FR breaks some E,
SmartASS breaks some E, etc.

Assume that public will accept any E not publicly broken.

Assume that we've figured out how to break another curve E.

Exploitability depends on public criteria for accepting E, P.

Extensive ECC literature:
Pollard rho breaks small E,
Pohlig–Hellman breaks most E,
MOV/FR breaks some E,
SmartASS breaks some E, etc.

Assume that public will accept any E not publicly broken.

Assume that we've figured out how to break another curve E.

Jerry standardizes this curve.
Alice and Bob use it.

Is first assumption plausible?

Would the public really accept
any curve chosen by Jerry
that survives these criteria?

Is first assumption plausible?

Would the public really accept
any curve chosen by Jerry
that survives these criteria?

Example showing plausibility:
Chinese OSCCA SM2 (2010)
includes algorithms and a curve.
The curve looks random;
survives these criteria;
has no other justification.

Is first assumption plausible?

Would the public really accept
any curve chosen by Jerry
that survives these criteria?

Example showing plausibility:
Chinese OSCCA SM2 (2010)
includes algorithms and a curve.
The curve looks random;
survives these criteria;
has no other justification.

More recent example:
French ANSSI FRP256V1 (2011).
Again no justification.

Maybe public is more demanding outside China and France: E must not be publicly broken, *and* Jerry must provide a "seed" s such that $E = H(s)$.

Maybe public is more demanding outside China and France: E must not be publicly broken, *and* Jerry must provide a "seed" s such that $E = H(s)$.

Examples: ANSI X9.62 (1999) "selecting an elliptic curve verifiably at random"; Certicom SEC 2 1.0 (2000) "verifiably random parameters offer some additional conservative features"—"parameters cannot be predetermined"; NIST FIPS 186-2 (2000); ANSI X9.63 (2001); Certicom SEC 2 2.0 (2010).

NIST defines curve E as
$y^2 = x^3 - 3x + b$ where
$b^2 c = -27$; c is a hash of s;
hash is SHA-1 concatenation.

NIST defines curve E as $y^2 = x^3 - 3x + b$ where $b^2c = -27$; c is a hash of s; hash is SHA-1 concatenation.

1999 Scott: "Consider now the possibility that one in a million of all curves have an exploitable structure that 'they' know about, but we don't. Then 'they' simply generate a million random seeds until they find one that generates one of 'their' curves. Then they get us to use them."

Optimized this computation on cluster of 41 GTX780 GPUs using H = Keccak. In 7 hours found "secure+twist-secure" b = 0x BADA55ECD8BBEAD3ADD6C534F92197DE B47FCEB9BE7E0E702A8D1DD56B5D0B0C mod NIST P-256.

Optimized this computation on cluster of 41 GTX780 GPUs using H = Keccak. In 7 hours found "secure+twist-secure" b = 0x

BADA55ECD8BBEAD3ADD6C534F92197DE
B47FCEB9BE7E0E702A8D1DD56B5D0B0C

mod NIST P-256.

Similarly found b = 0x

BADA55ECFD9CA54C0738B8A6FB8CF4CC
F84E916D83D6DA1B78B622351E11AB4E

mod NIST P-224; and b = 0x

BADA55EC3BE2AD1F9EEEA5881ECF95BB
F3AC392526F01D4CD13E684C63A17CC4
D5F271642AD83899113817A61006413D

mod NIST P-384.

Maybe in some countries
the public is more demanding.

Maybe in some countries
the public is more demanding.

Brainpool standard (2005):
"The choice of the seeds
from which the [NIST] curve
parameters have been derived is
not motivated leaving an essential
part of the security analysis open.
. . . **Verifiably pseudo-random.**
The [Brainpool] curves shall be
generated in a pseudo-random
manner using seeds that are
generated in a systematic and
comprehensive way."

```
import hashlib
def hash(seed): h = hashlib.sha1(); h.update(seed); return h.digest()
seedbytes = 20

p = 0xD7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
k = GF(p); R.<x> = k[]

def secure(A,B):
  if k(B).is_square(): return False
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n < p and n.is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1)

def int2str(seed,bytes):
  return ''.join([chr((seed//256^i)%256) for i in reversed(range(bytes))])

def str2int(seed):
  return Integer(seed.encode('hex'),16)

def update(seed):
  return int2str(str2int(seed) + 1,len(seed))

def fullhash(seed):
  return str2int(hash(seed) + hash(update(seed))) % 2^223

def real2str(seed,bytes):
  return int2str(Integer(floor(RealField(8*bytes+8)(seed)*256^bytes)),bytes)

nums = real2str(exp(1)/16,7*seedbytes)
S = nums[2*seedbytes:3*seedbytes]
while True:
  A = fullhash(S)
  if not (k(A)*x^4+3).roots(): S = update(S); continue
  S = update(S)
  B = fullhash(S)
  if not secure(A,B): S = update(S); continue
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()
  break
```

We carefully implemented the curve-generation procedure from the Brainpool standard. Previous slide: 224-bit procedure.

Output of this procedure:

```
p  D7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
A  2B98B906DC245F2916C03A2F953EA9AE565C3253E8AEC4BFE84C659E
B  68AEC4BFE84C659EBB8B81DC39355A2EBFA3870D98976FA2F17D2D8D
```

We carefully implemented the curve-generation procedure from the Brainpool standard. Previous slide: 224-bit procedure.

Output of this procedure:

```
p  D7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
A  2B98B906DC245F2916C03A2F953EA9AE565C3253E8AEC4BFE84C659E
B  68AEC4BFE84C659EBB8B81DC39355A2EBFA3870D98976FA2F17D2D8D
```

The standard 224-bit Brainpool curve **is not the same curve**:

```
p  D7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
A  68A5E62CA9CE6C1C299803A6C1530B514E182AD8B0042A59CAD29F43
B  2580F63CCFE44138870713B1A92369E33E2135D266DBB372386C400B
```

We carefully implemented the curve-generation procedure from the Brainpool standard. Previous slide: 224-bit procedure.

Output of this procedure:

```
p  D7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
A  2B98B906DC245F2916C03A2F953EA9AE565C3253E8AEC4BFE84C659E
B  68AEC4BFE84C659EBB8B81DC39355A2EBFA3870D98976FA2F17D2D8D
```

The standard 224-bit Brainpool curve **is not the same curve**:

```
p  D7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
A  68A5E62CA9CE6C1C299803A6C1530B514E182AD8B0042A59CAD29F43
B  2580F63CCFE44138870713B1A92369E33E2135D266DBB372386C400B
```

Next slide: a procedure that **does** generate the standard Brainpool curve.

```
import hashlib
def hash(seed): h = hashlib.sha1(); h.update(seed); return h.digest()
seedbytes = 20

p = 0xD7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
k = GF(p); R.<x> = k[]

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n < p and n.is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1)

def int2str(seed,bytes):
  return ''.join([chr((seed//256^i)%256) for i in reversed(range(bytes))])

def str2int(seed):
  return Integer(seed.encode('hex'),16)

def update(seed):
  return int2str(str2int(seed) + 1,len(seed))

def fullhash(seed):
  return str2int(hash(seed) + hash(update(seed))) % 2^223

def real2str(seed,bytes):
  return int2str(Integer(floor(RealField(8*bytes+8)(seed)*256^bytes)),bytes)

nums = real2str(exp(1)/16,7*seedbytes)
S = nums[2*seedbytes:3*seedbytes]
while True:
  A = fullhash(S)
  if not (k(A)*x^4+3).roots(): S = update(S); continue
  while True:
    S = update(S)
    B = fullhash(S)
    if not k(B).is_square(): break
  if not secure(A,B): S = update(S); continue
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()
  break
```

Did Brainpool check before publication? After publication? Did they know before 2015?

Brainpool procedure is advertised as "systematic", "comprehensive", "completely transparent", etc. Surely we can say the same for *both* procedures.

Can quietly manipulate choice to take the weaker procedure.

Did Brainpool check before publication? After publication? Did they know before 2015?

Brainpool procedure is advertised as “systematic”, “comprehensive”, “completely transparent”, etc. Surely we can say the same for *both* procedures.

Can quietly manipulate choice to take the weaker procedure.

Interesting Brainpool quote: “It is envisioned to provide additional curves on a regular basis.”

We made a new 224-bit curve using standard NIST P-224 prime.

To avoid Brainpool's complications of concatenating hash outputs: We upgraded from SHA-1 to state-of-the-art maximum-security SHA3-512. Also upgraded to requiring maximum twist security.

Brainpool uses $\exp(1) = e$ and $\arctan(1) = \pi/4$, and MD5 uses $\sin(1)$, so we used $\cos(1)$. We also used much simpler pattern of searching for seeds.

```
import simplesha3
hash = simplesha3.sha3512

p = 2^224 - 2^96 + 1
k = GF(p)
seedbytes = 20

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

def int2str(seed,bytes):
  return ''.join([chr((seed//256^i)%256) for i in reversed(range(bytes))])

def str2int(seed):
  return Integer(seed.encode('hex'),16)

def complement(seed):
  return ''.join([chr(255-ord(s)) for s in seed])

def real2str(seed,bytes):
  return int2str(Integer(RealField(8*bytes)(seed)*256^bytes),bytes)

sizeofint = 4
nums = real2str(cos(1),seedbytes - sizeofint)
for counter in xrange(0,256^sizeofint):
  S = int2str(counter,sizeofint) + nums
  T = complement(S)
  A = str2int(hash(S))
  B = str2int(hash(T))
  if secure(A,B):
    print 'p',hex(p).upper()
    print 'A',hex(A).upper()
    print 'B',hex(B).upper()
    break
```

Output: `7144BA12CE8A0C3BEFA053EDBADA55...`

Output: `7144BA12CE8A0C3BEFA053EDBADA55...`

We actually generated >1000000 curves for this prime, each having a Brainpool-like explanation, even without complicating hashing, seed search, etc.; e.g., `BADA55-VPR2-224` uses exp(1).

Output: `7144BA12CE8A0C3BEFA053EDBADA55...`

We actually generated $>$1000000 curves for this prime, each having a Brainpool-like explanation, even without complicating hashing, seed search, etc.; e.g., `BADA55-VPR2-224` uses exp(1).

See `bada55.cr.yp.to` for much more: full paper; scripts; detailed Brainpool analysis; manipulating "minimal" primes and curves (Microsoft "NUMS"); manipulating security criteria.