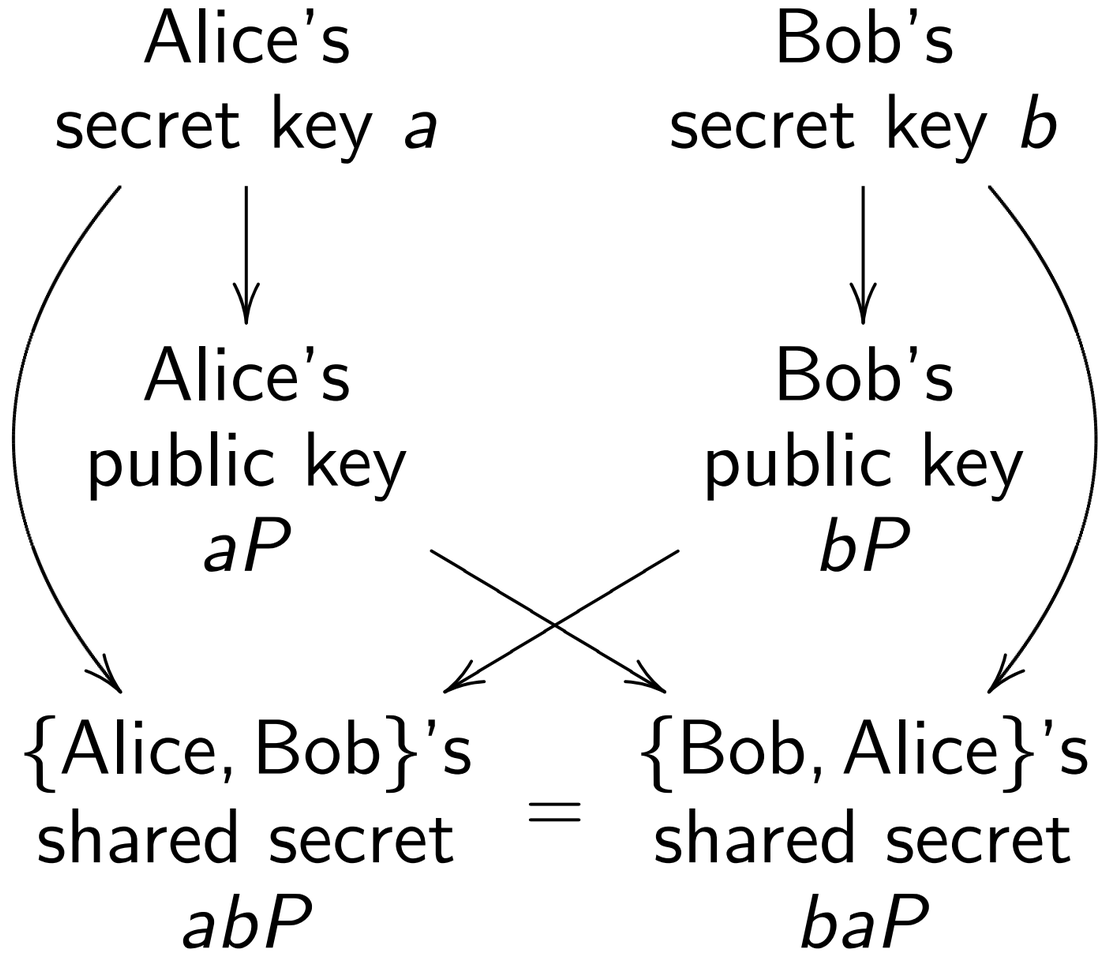
How to
manipulate curve standards:
a white paper for the black hat

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:

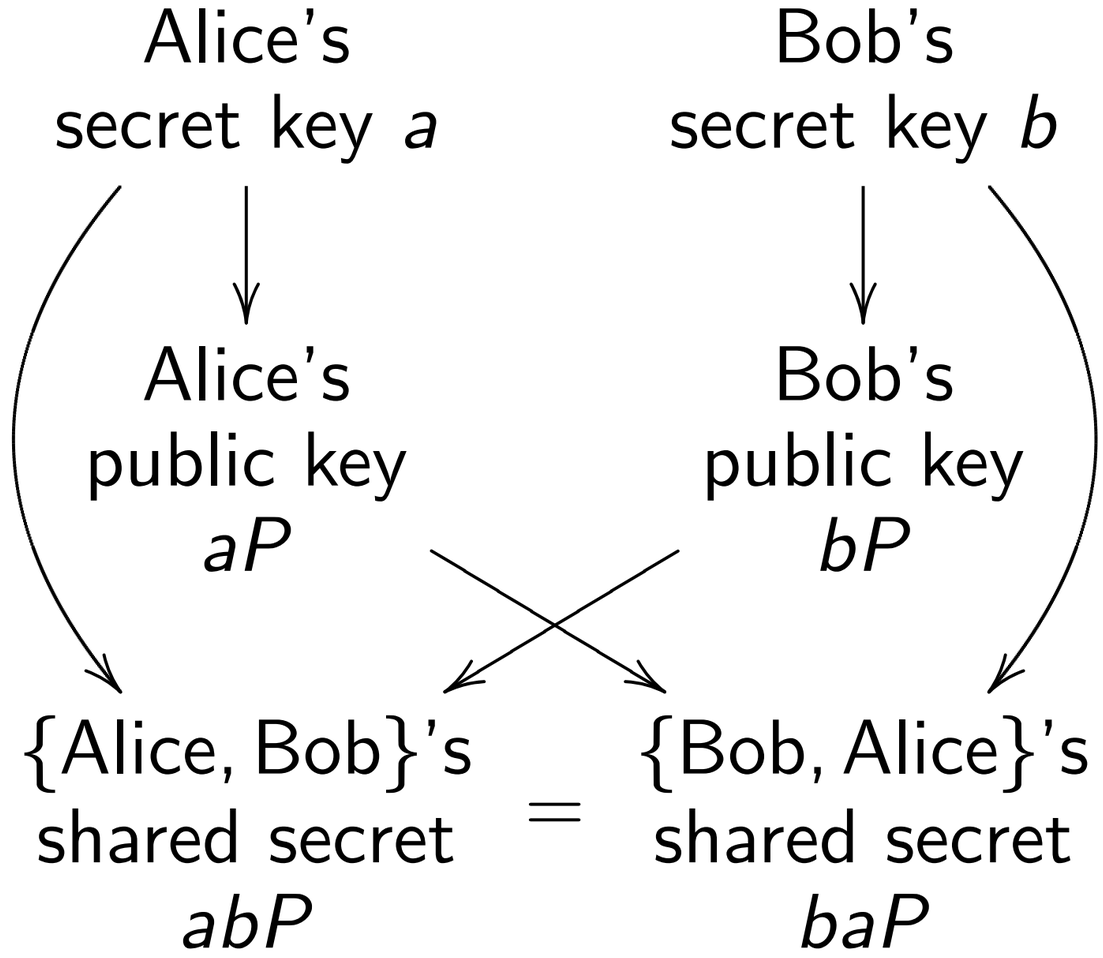


How to
manipulate curve standards:
a white paper for the black hat

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:



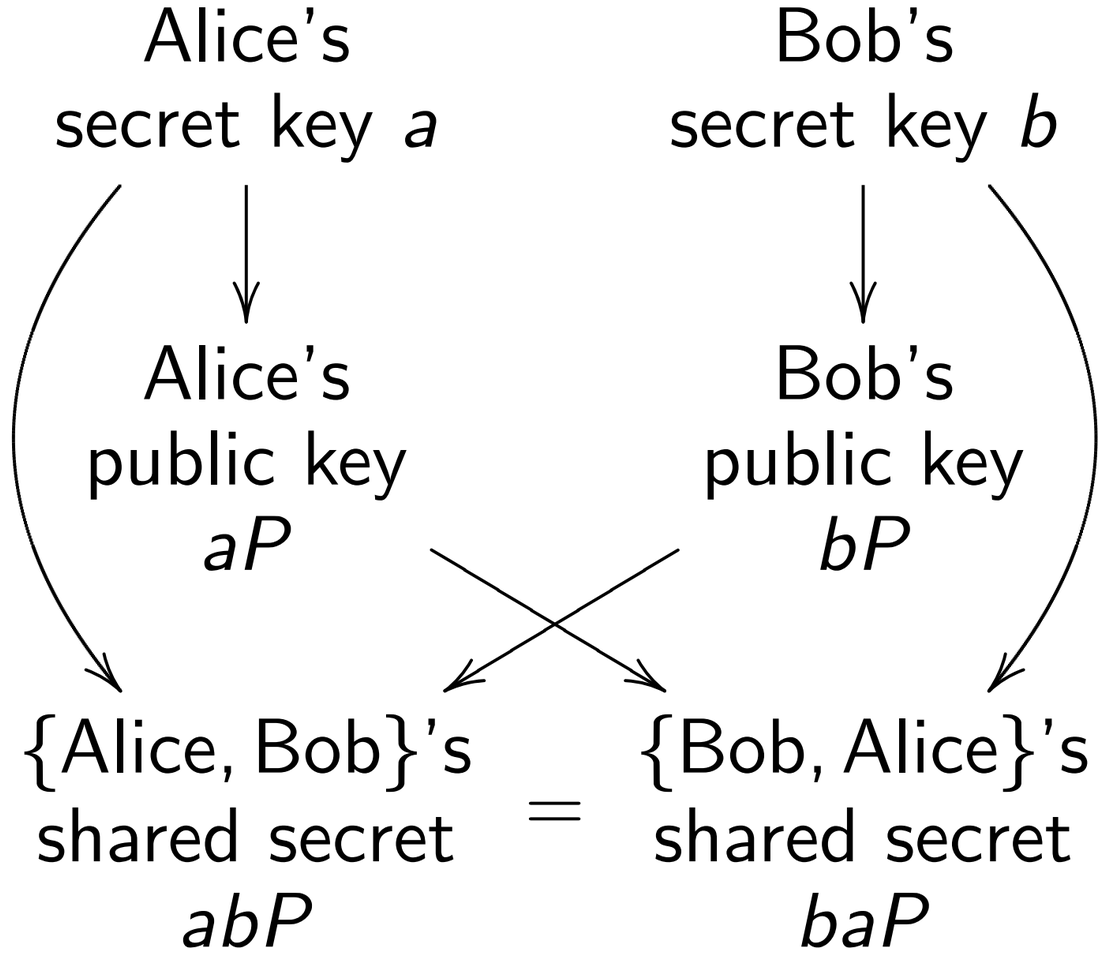
Security depends on choice of E.

ate curve standards:
paper for the black hat

. Bernstein
hou
hok Chuengsatiansup
Hülsing
mbooij
ange
Niederhagen
e van Vredendaal

.cr.yp.to

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



Security depends on choice of E.

Alic
secret
Alic
public
aP
{Alice,
shared
ab

standards:

the black hat

n

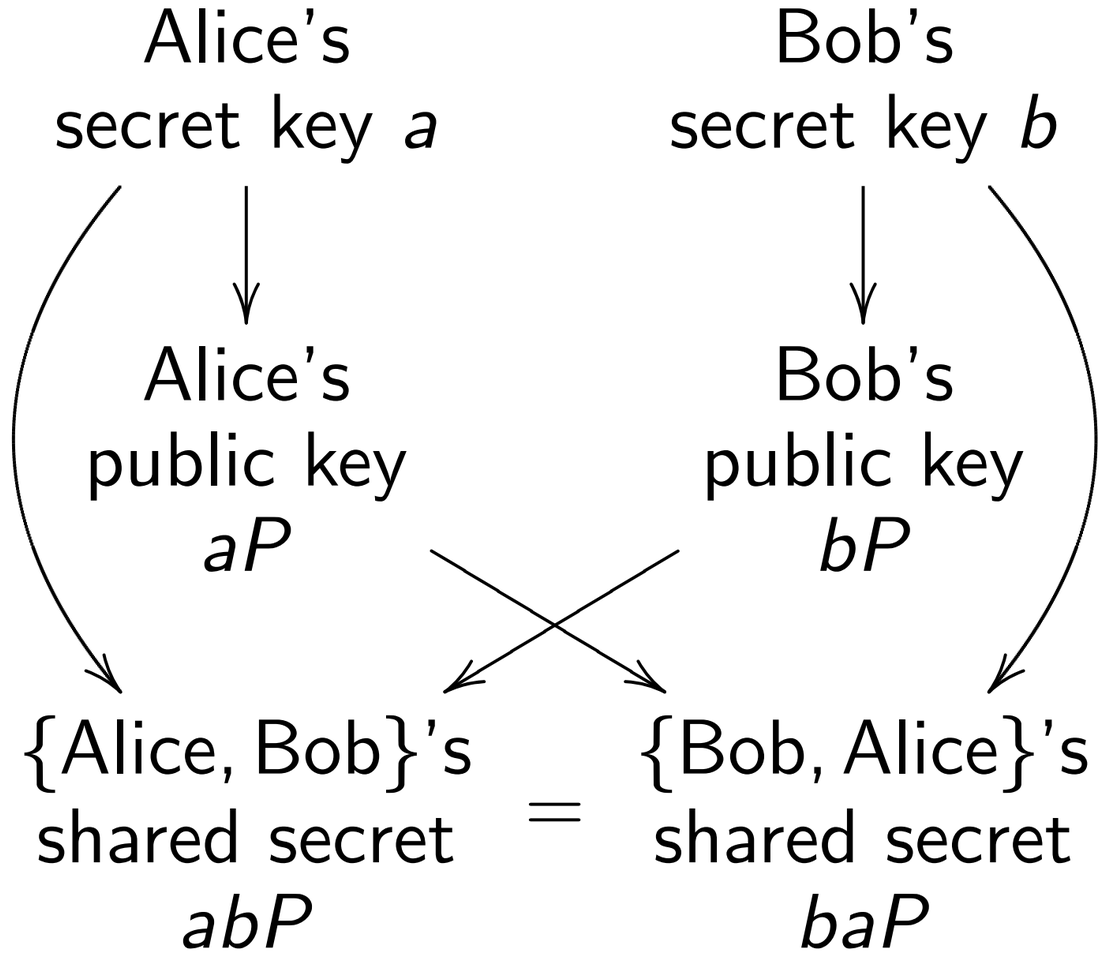
gsatiansup

en

lendaal

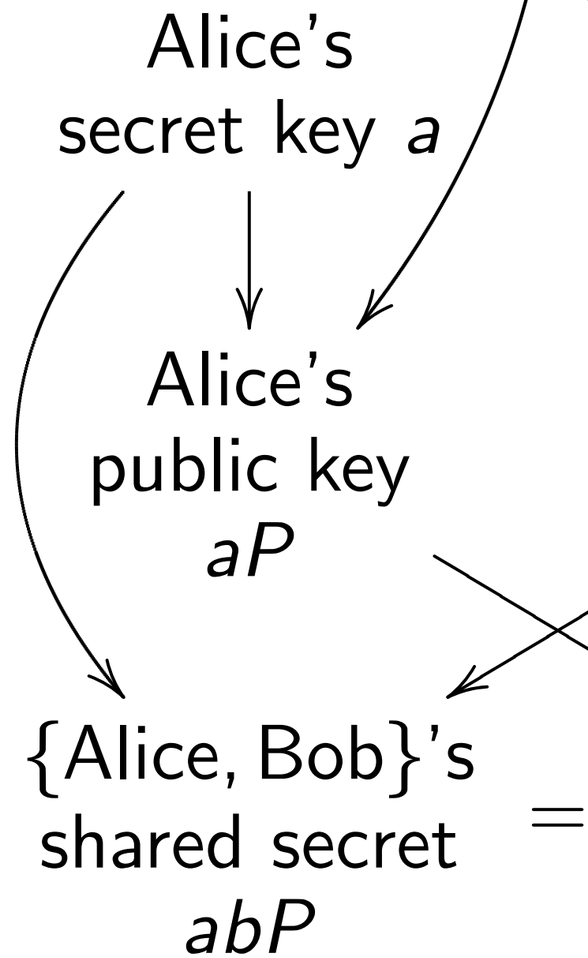
o

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



Security depends on choice of E.

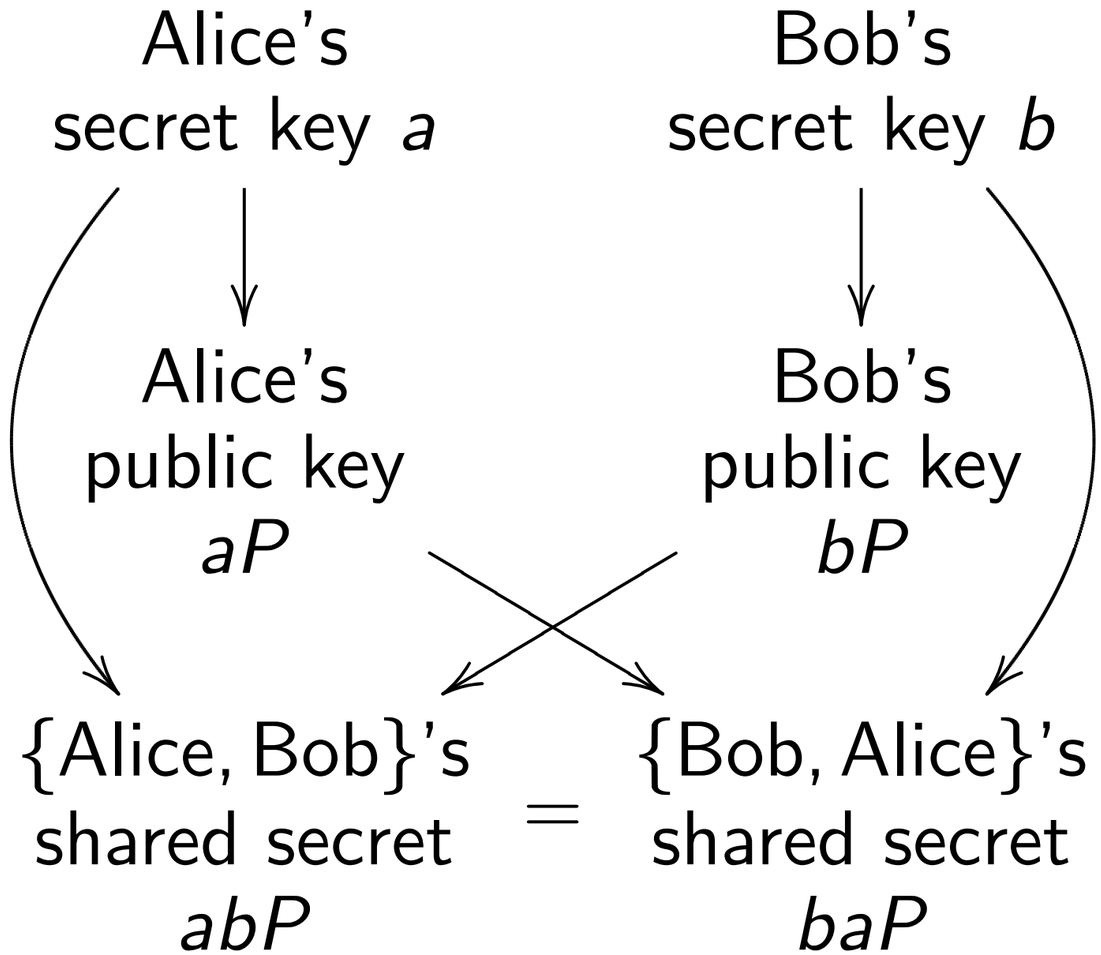
Our partne
choice o



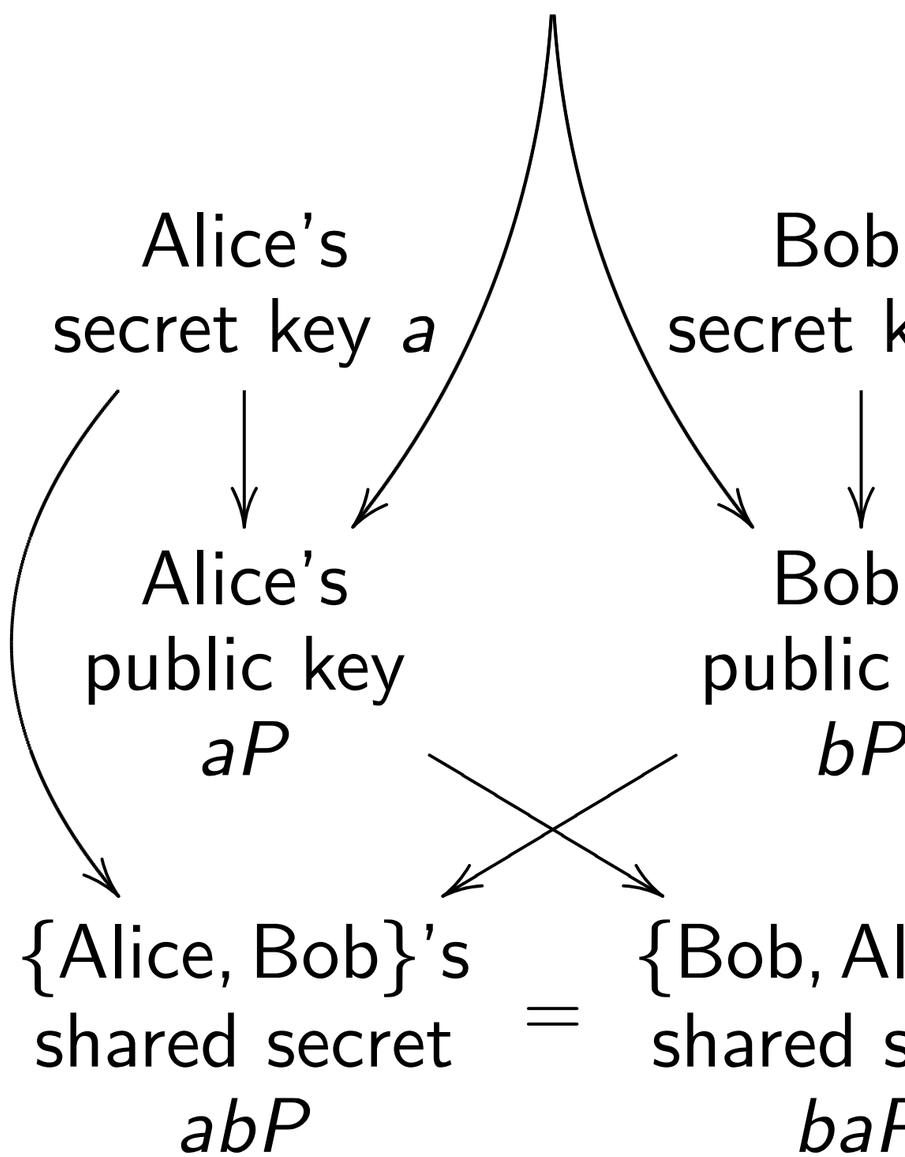
hat

p

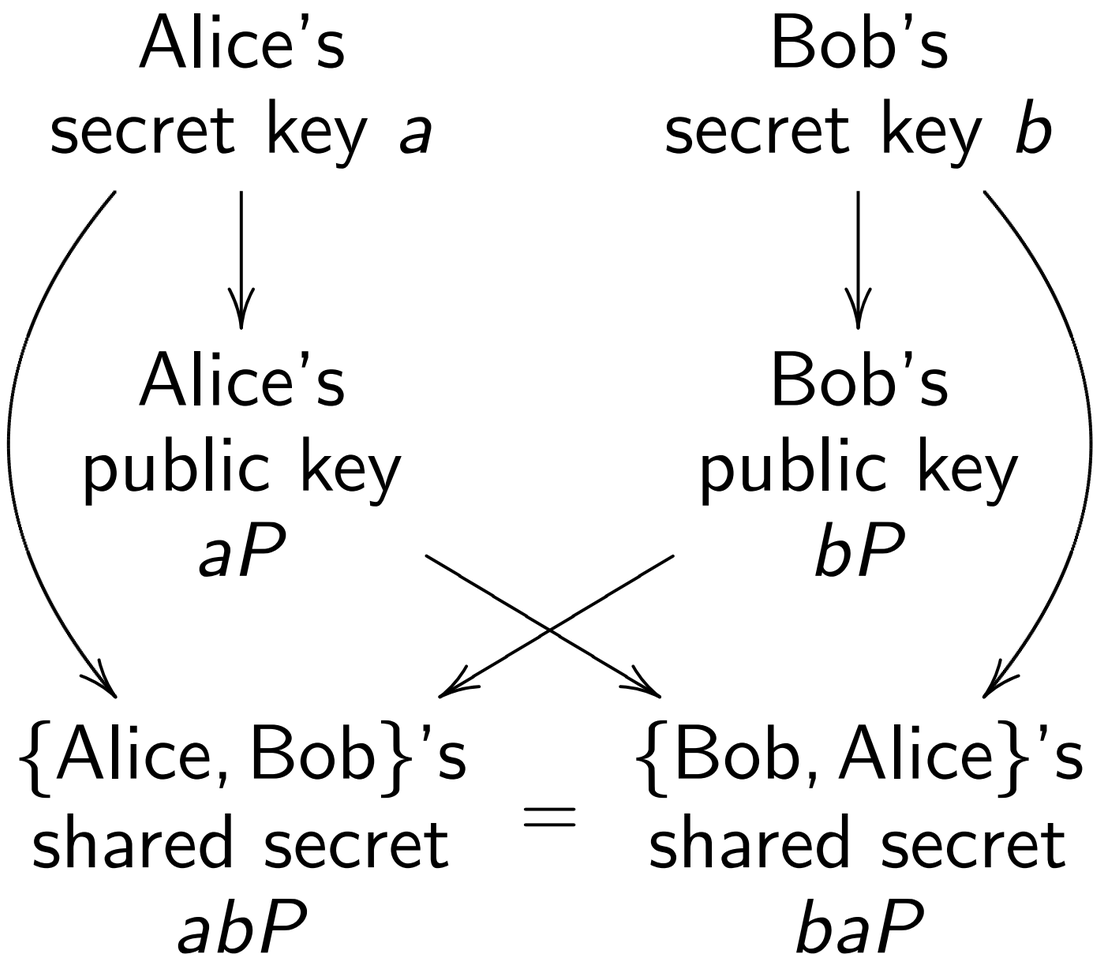
Textbook key exchange
using standard point P
on a standard elliptic curve E:



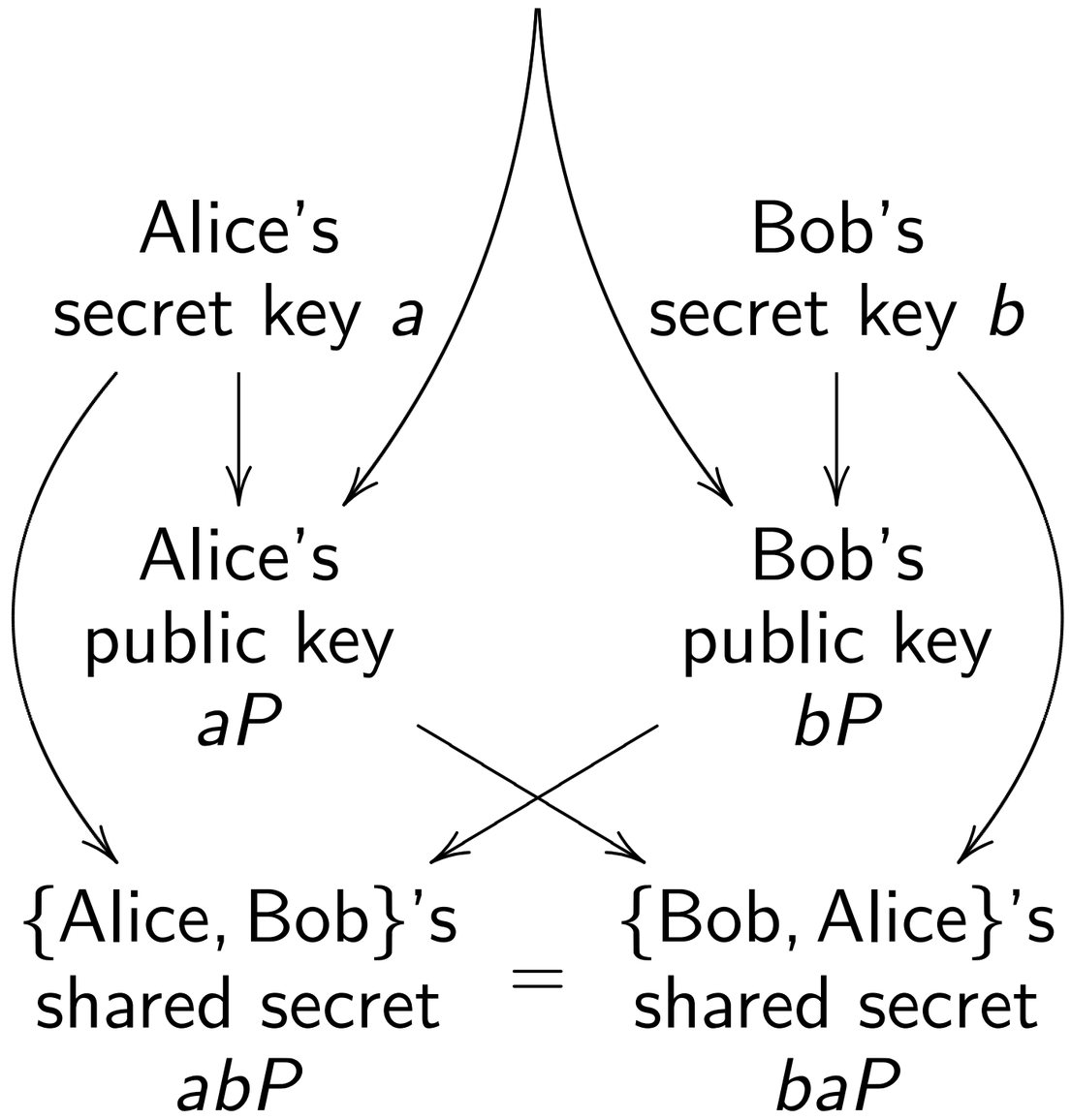
Security depends on choice of E.



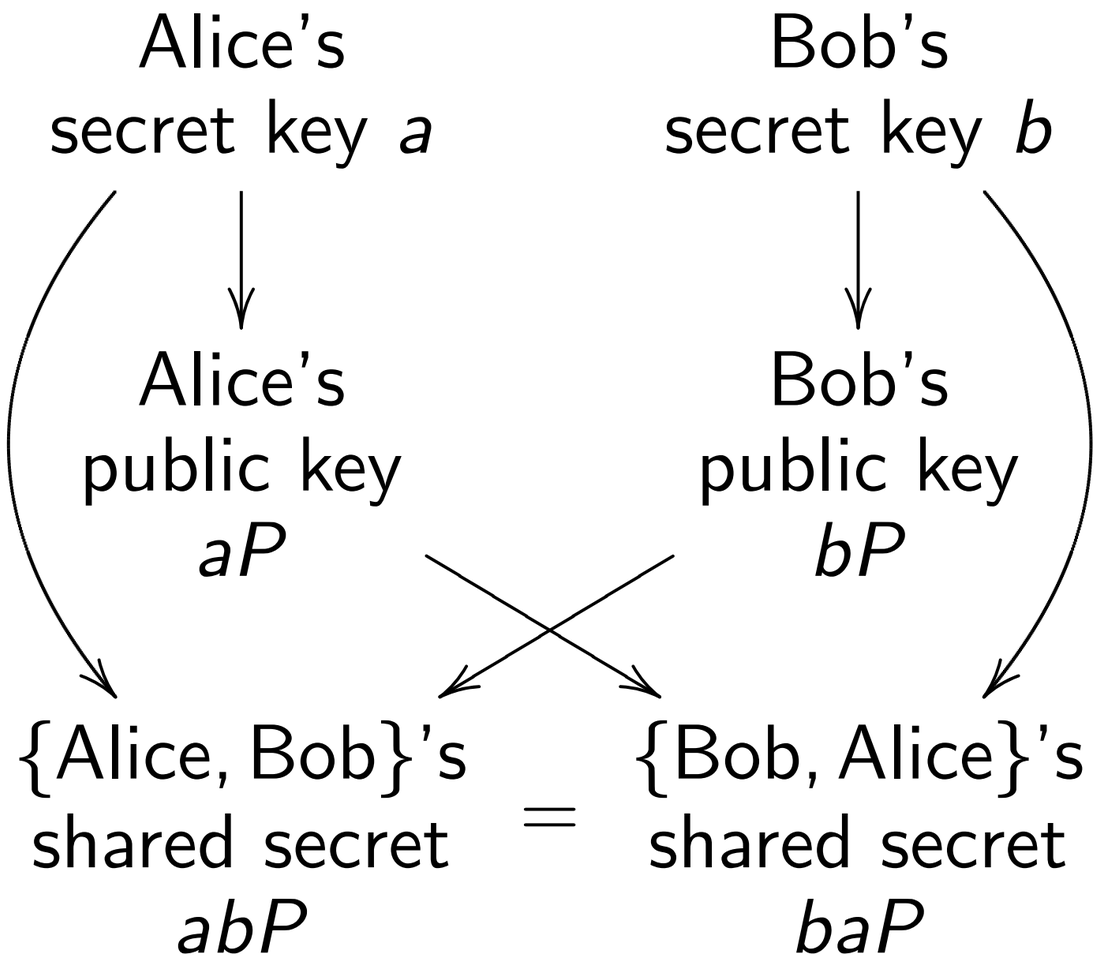
Textbook key exchange
using standard point P
on a standard elliptic curve E:



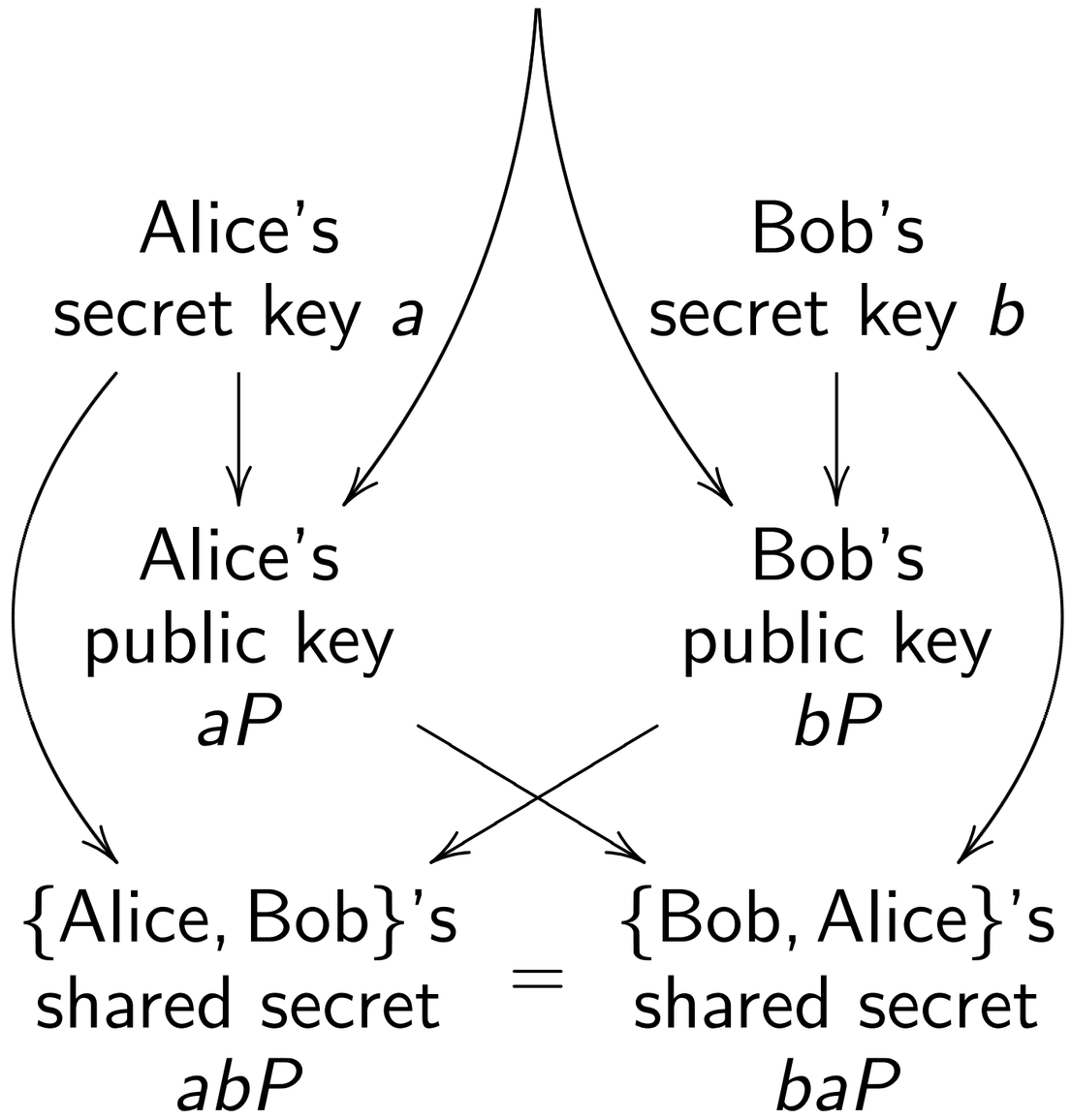
Security depends on choice of E.



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



Security depends on choice of E.

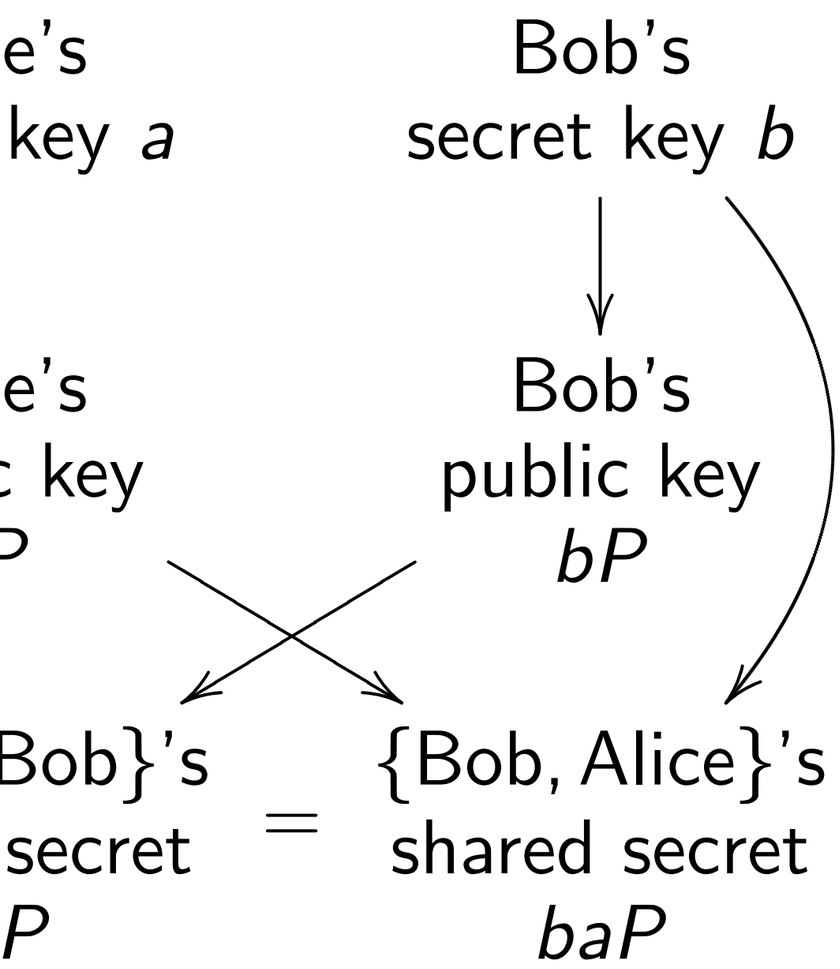


Can we exploit this picture?

k key exchange

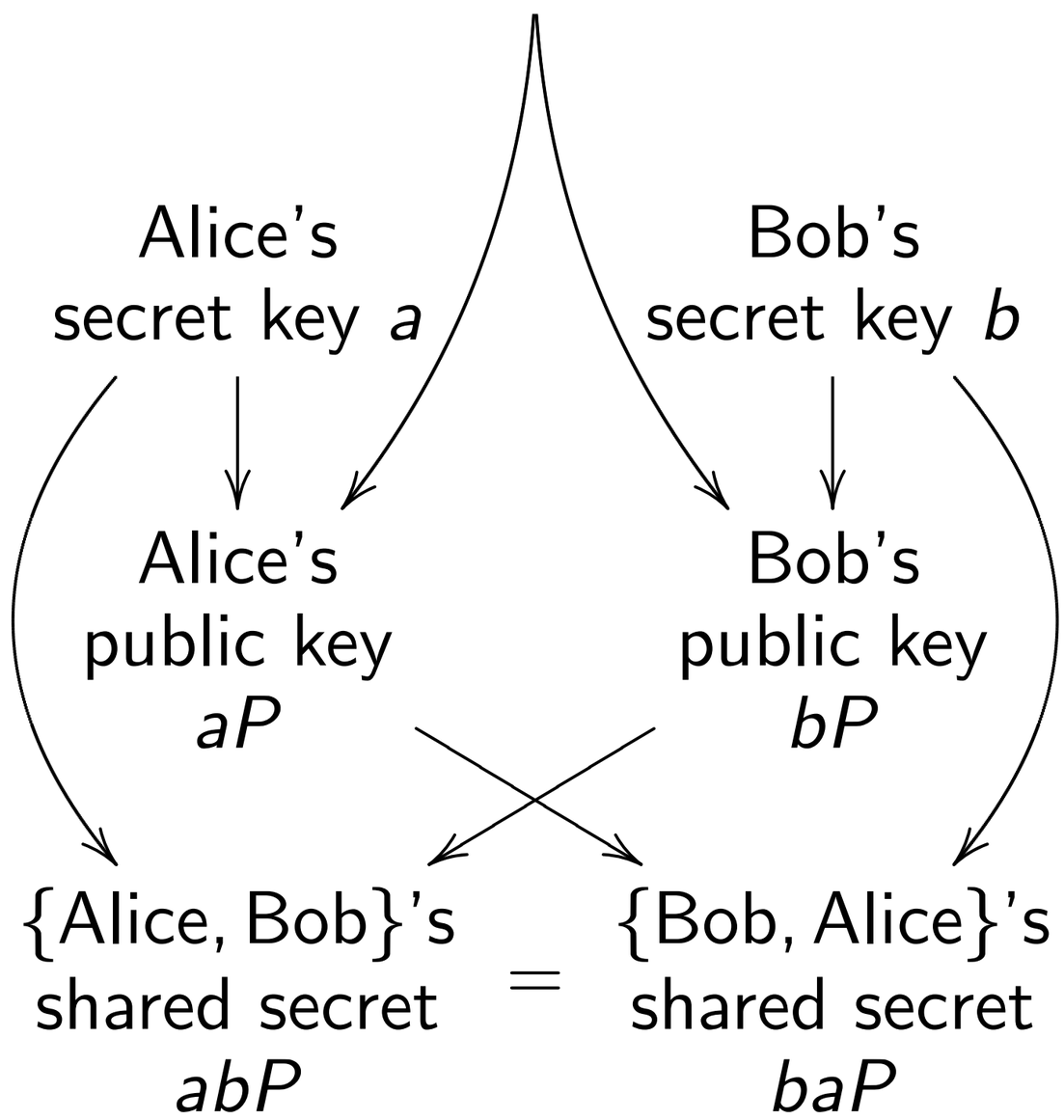
andard point P

ndard elliptic curve E:



depends on choice of E.

Our partner Jerry's
choice of E, P



Can we exploit this picture?

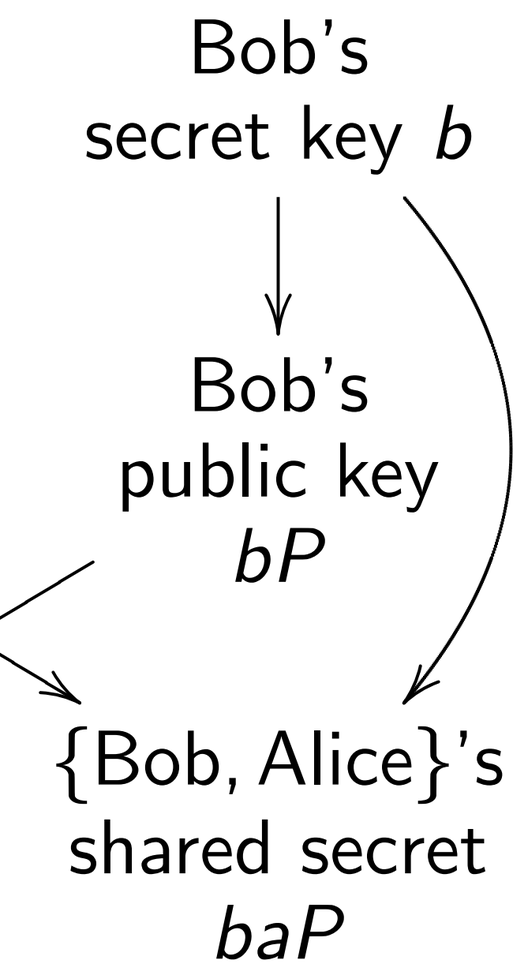
Exploita

public cr

nange

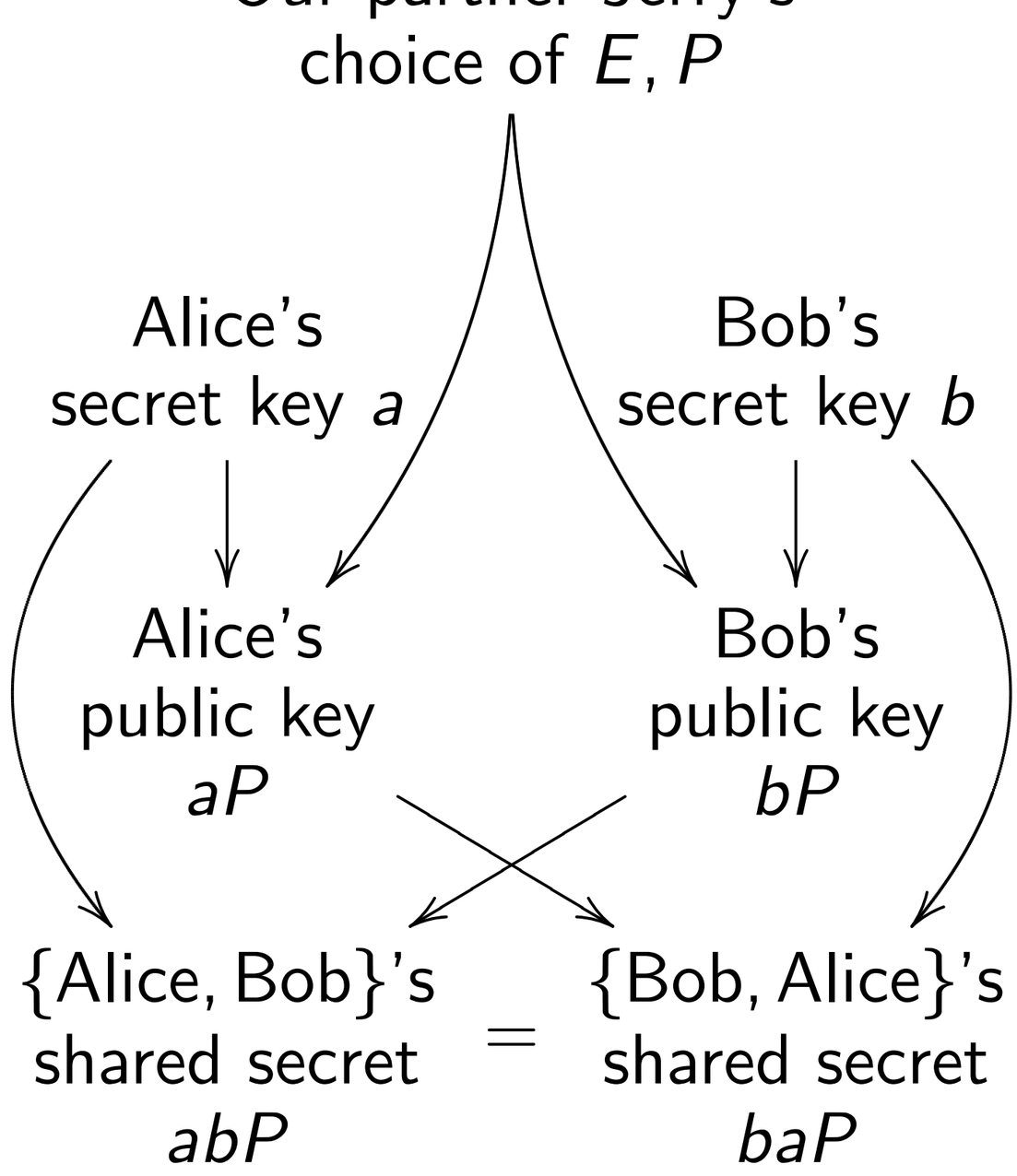
int P

otic curve E:



on choice of E.

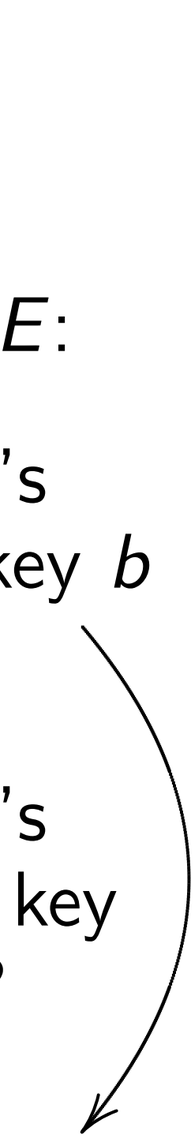
Our partner Jerry's
choice of E, P

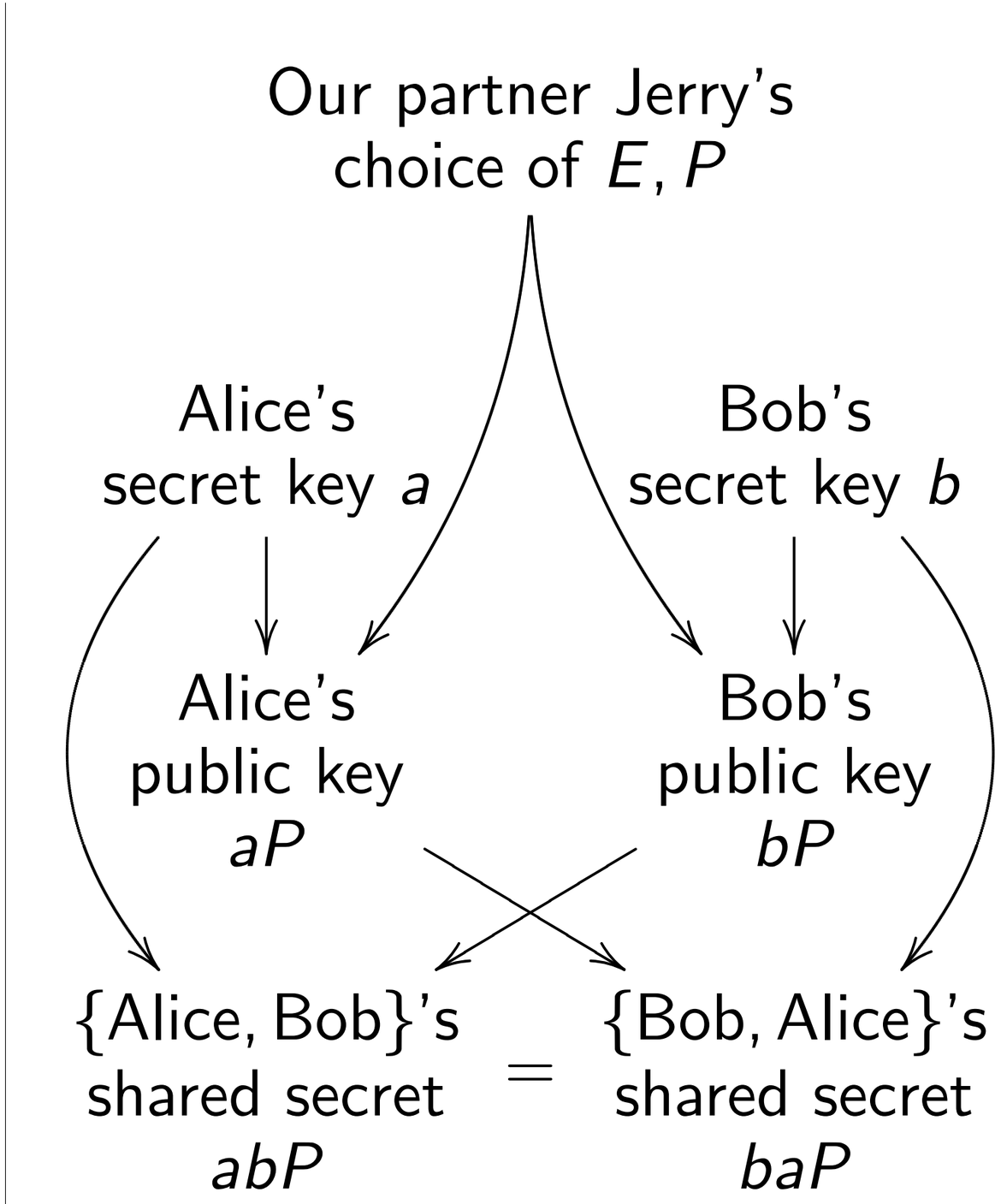


Can we exploit this picture?

Exploitability depe

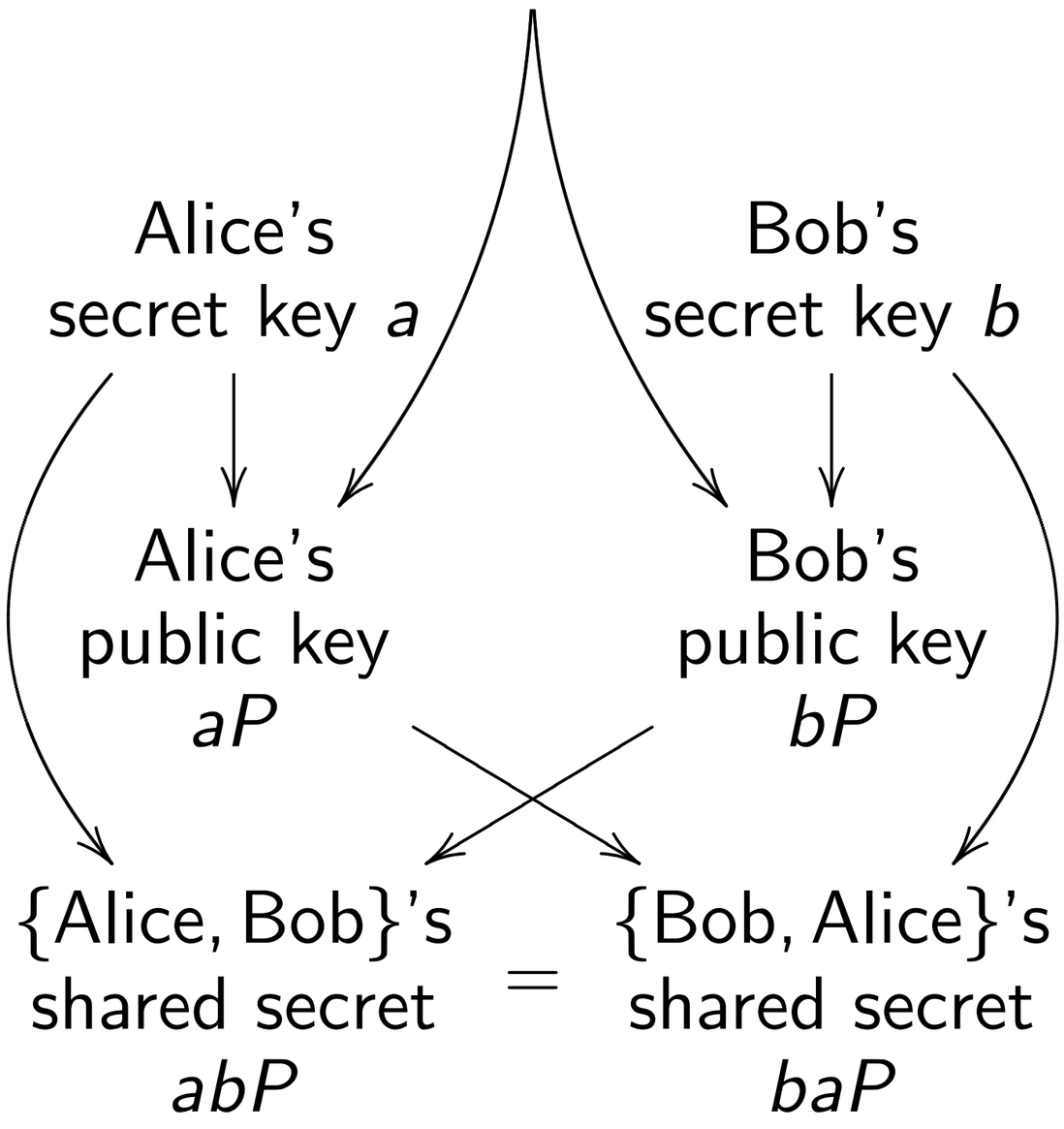
public criteria for a





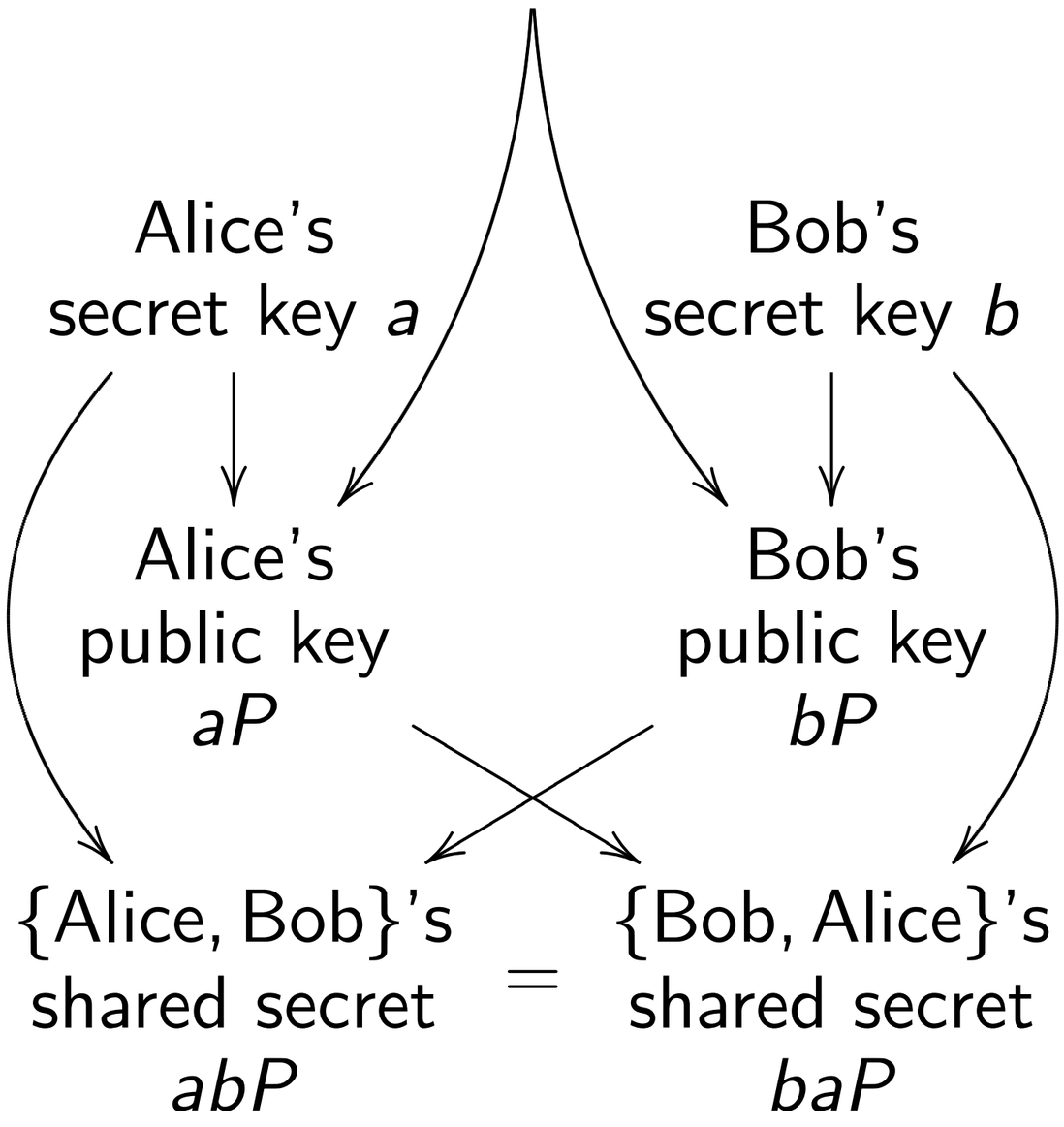
Can we exploit this picture?

Exploitability depends on public criteria for accepting



Can we exploit this picture?

Exploitability depends on public criteria for accepting E, P.

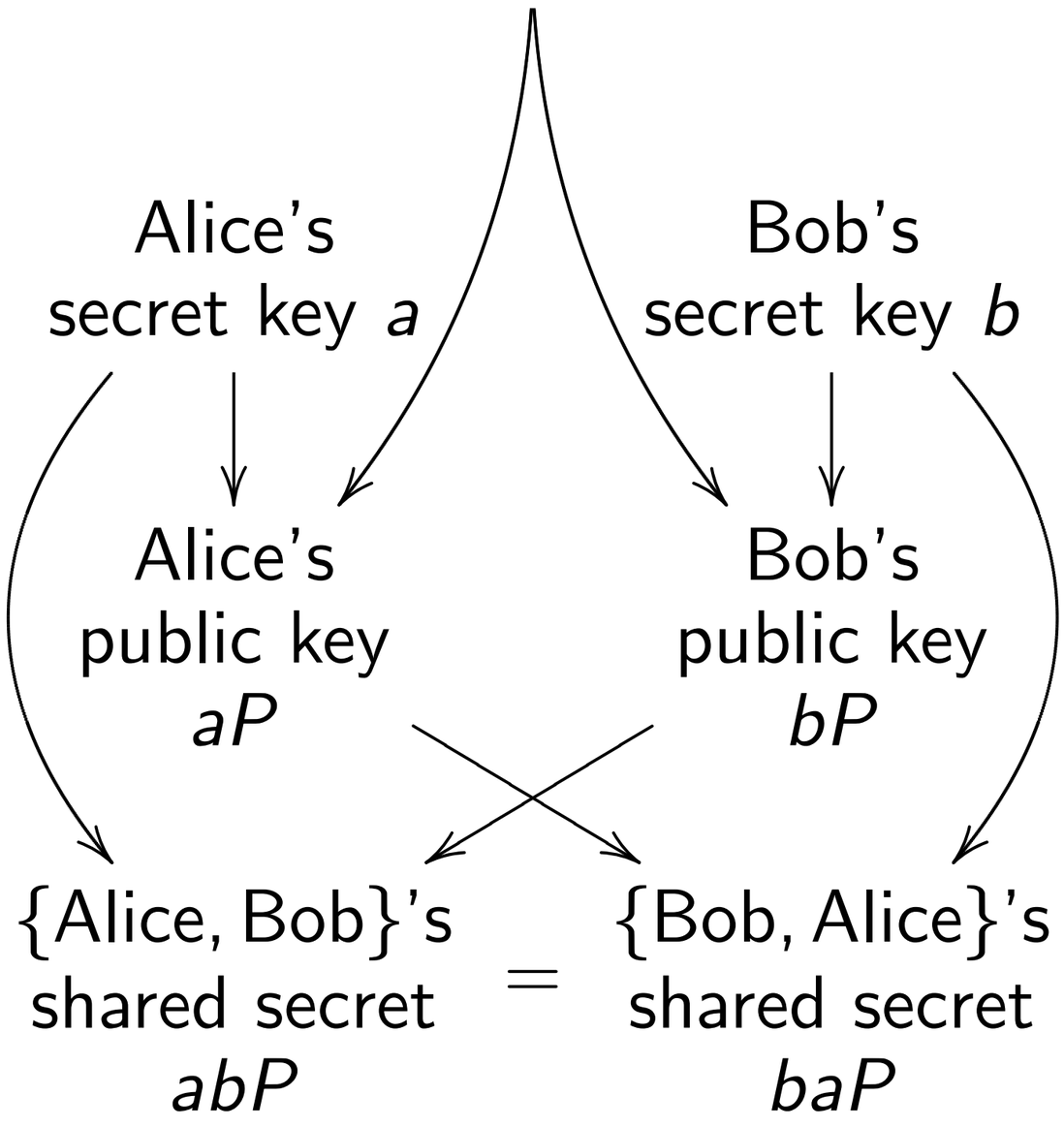


Can we exploit this picture?

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.



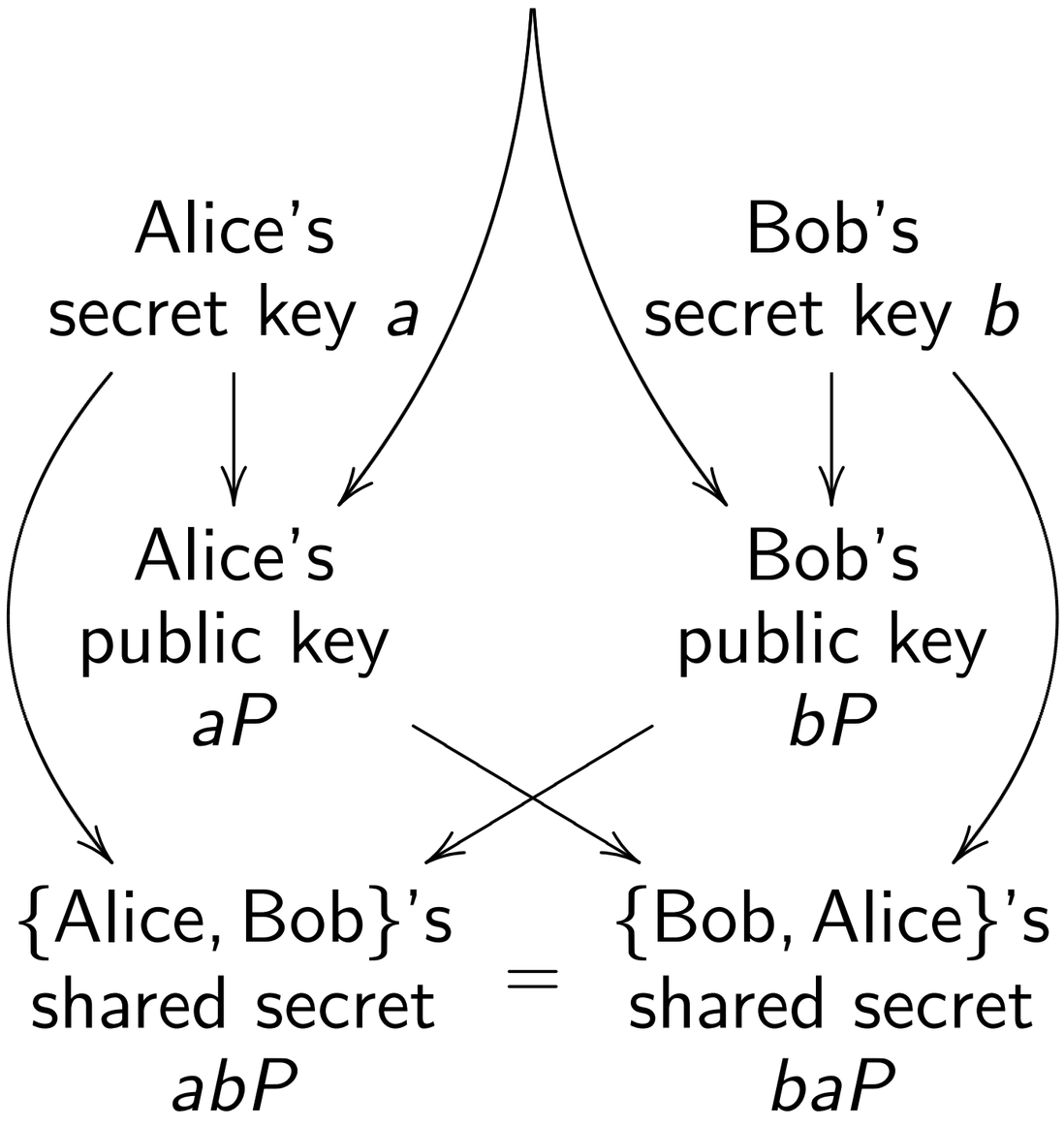
Can we exploit this picture?

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.



Can we exploit this picture?

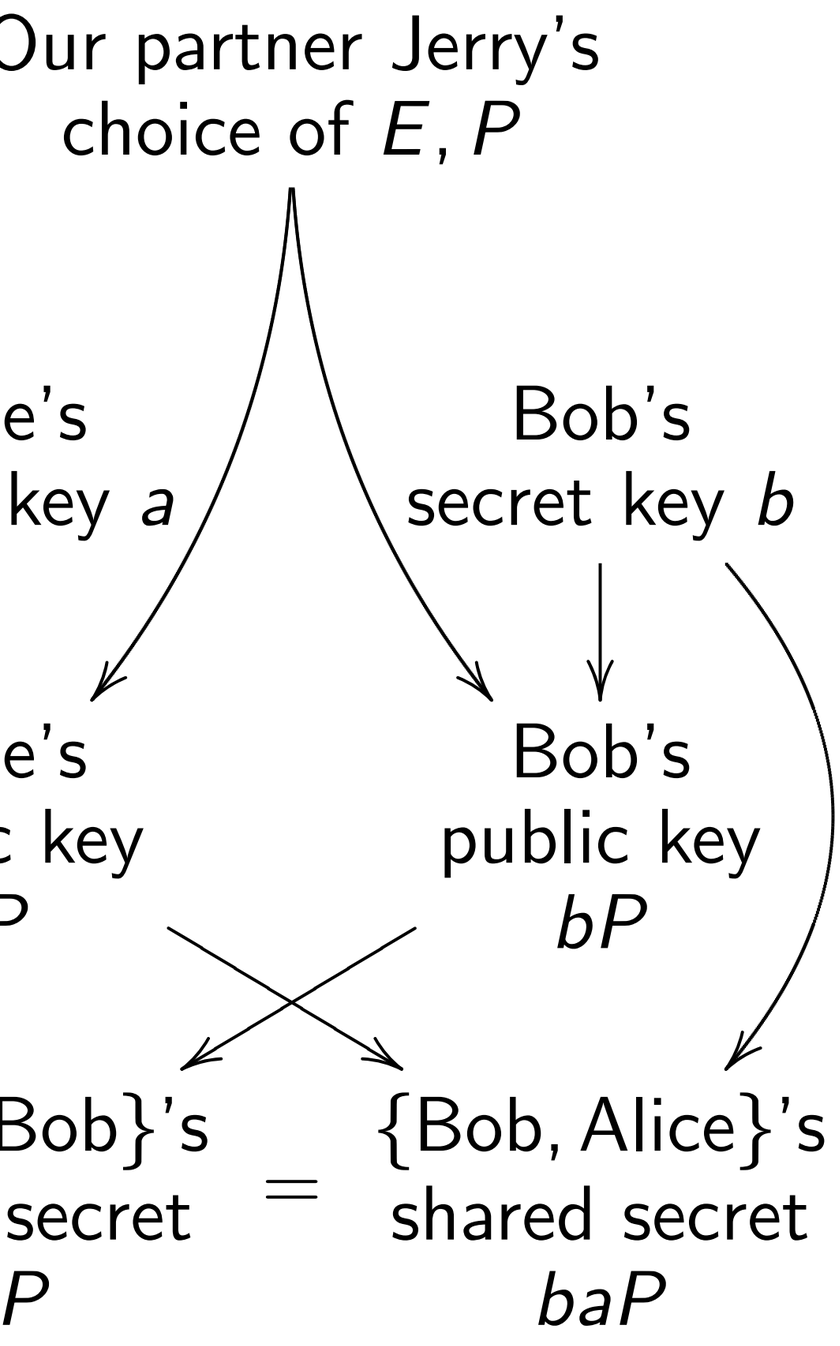
Exploitability depends on
public criteria for accepting E, P.

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.



exploit this picture?

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.

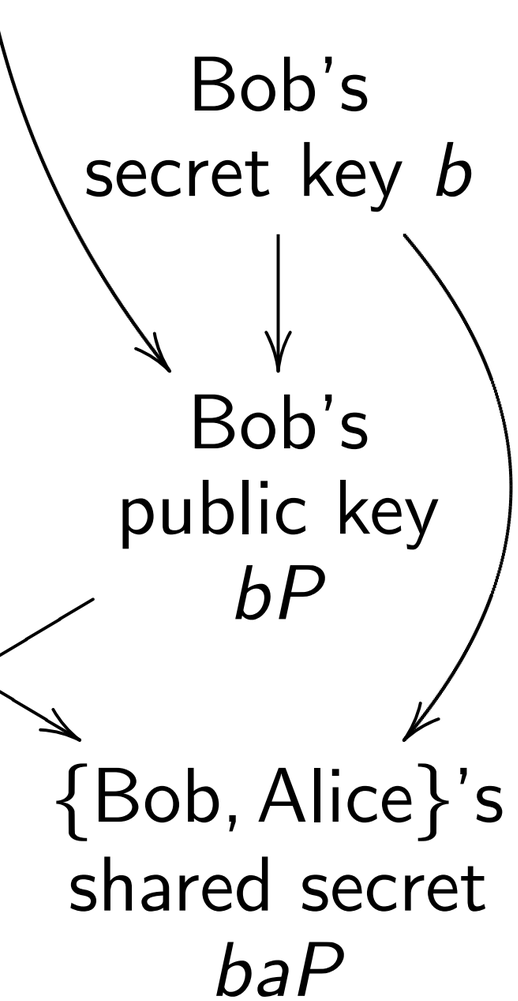
Assume that we've figured out
how to break another curve E.

Jerry standardizes this curve.
Alice and Bob use it.

Is first a

Would t
any curv
that surv

er Jerry's
f E, P



s picture?

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

Would the public

any curve chosen

that survives these

's
key b

's
key

ice}'s
secret
?

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 acce
any curve chosen by Jerry
that survives these criteria?

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?

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?

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

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?

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.

bility depends on
riteria for accepting E, P.

e ECC literature:
rho breaks small E,
Hellman breaks most E,
R breaks some E,
SS breaks some E, etc.

that public will accept
ot publicly broken.

that we've figured out
break another curve E.

andardizes this curve.
d Bob use it.

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 p
outside
E must
and Jerr
"seed" *s*

ends on
accepting E, P.

erature:
small E,
reaks most E,
some E,
some E, etc.

c will accept
broken.

e figured out
her curve E.

this curve.
it.

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 m
outside China and
E must not be pu
and Jerry must pr
"seed" s such that

E, P.

E,

tc.

ept

out

E.

.

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 dema
outside China and France:
E must not be publicly brok
and Jerry must provide a
"seed" s such that $E = H(s$

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 $E = H(s)$.

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 $E = H(s)$.

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).

ssumption plausible?

he public really accept
ve chosen by Jerry
vives these criteria?

e showing plausibility:
OSCCA SM2 (2010)
algorithms and a curve.
ve looks random;
these criteria;
other justification.

cent example:
ANSSI FRP256V1 (2011).
o 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 $E = H(s)$.

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 de
$y^2 = x^3$
$b^2c = -$
hash is S

plausible?

really accept
by Jerry
e criteria?

plausibility:
M2 (2010)
s and a curve.
andom;
eria;
fication.

ple:
P256V1 (2011).
tion.

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 curv
$y^2 = x^3 - 3x + b$
$b^2c = -27$; c is a
hash is SHA-1 co

?

ept

y:

)

urve.

2011).

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 $b^2c = -27$; c is a hash of s hash is SHA-1 concatenation

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 $b^2c = -27$; c is a hash of s; hash is SHA-1 concatenation.

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 $b^2c = -27$; 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."

ublic is more demanding
China and France:
not be publicly broken,
y must provide a
such that $E = H(s)$.

es: ANSI X9.62 (1999)
g an elliptic curve
y at random"; Certicom
.0 (2000) "verifiably
parameters offer
ditional conservative
"—"parameters cannot
etermined"; NIST FIPS
2000); ANSI X9.63 (2001);
n SEC 2 2.0 (2010).

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."

Optimiz
cluster c
H = Ke
"secure-
BADA55E
B47FCEB
mod NI

ore demanding
France:
blicly broken,
ovide a
t $E = H(s)$.

K9.62 (1999)
tic curve
m"; Certicom
"verifiably
rs offer
onservative
neters cannot
; NIST FIPS
SI X9.63 (2001);
.0 (2010).

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 co
cluster of 41 GTX
H = Keccak. In
"secure+twist-sec
BADA55ECD8BBEAD3
B47FCEB9BE7E0E70
mod NIST P-256.

nding

ken,

s).

99)

com

y

e

not

PS

2001);

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 cluster of 41 GTX780 GPUs H = Keccak. In 7 hours fo "secure+twist-secure" $b =$ BADA55ECD8BBEAD3ADD6C534F B47FCEB9BE7E0E702A8D1DD56 mod NIST P-256.

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.

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.

Similarly found $b =$ 0x

BADA55ECFD9CA54C0738B8A6FB8CF4CC

F84E916D83D6DA1B78B622351E11AB4E

mod NIST P-224; and $b =$ 0x

BADA55EC3BE2AD1F9EEEA5881ECF95BB

F3AC392526F01D4CD13E684C63A17CC4

D5F271642AD8389913817A61006413D

mod NIST P-384.

efines curve E as
$-3x+b$ where
-27; c is a hash of s;
SHA-1 concatenation.

ott: "Consider now the
ty that one in a million
rves have an exploitable
e that 'they' know about,
don't. Then 'they' simply
e a million random seeds
ey find one that generates
their' curves. Then they
o 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.
Similarly found $b =$ 0x
BADA55ECFD9CA54C0738B8A6FB8CF4CC
F84E916D83D6DA1B78B622351E11AB4E
mod NIST P-224; and $b =$ 0x
BADA55EC3BE2AD1F9EEEA5881ECF95BB
F3AC392526F01D4CD13E684C63A17CC4
D5F271642AD8389911381 7A61006413D
mod NIST P-384.

Maybe i
the publ

e E as
where
hash of s;
ncatenation.

sider now the
e in a million
an exploitable
ey' know about,
en 'they' simply
random seeds
e that generates
es. Then they
n."

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
D5F271642AD8389911381 7A61006413D
mod NIST P-384.

Maybe in some co
the public is more

;
n.

the
ion
able
bout,
imply
eeds
erates
they

Optimized this computation on cluster of 41 GTX780 GPUs using H = Keccak. In 7 hours found "secure+twist-secure" $b = \texttt{0x}$
BADA55ECD8BBEAD3ADD6C534F92197DE
B47FCEB9BE7E0E702A8D1DD56B5D0B0C
mod NIST P-256.
Similarly found $b = \texttt{0x}$
BADA55ECFD9CA54C0738B8A6FB8CF4CC
F84E916D83D6DA1B78B622351E11AB4E
mod NIST P-224; and $b = \texttt{0x}$
BADA55EC3BE2AD1F9EEEA5881ECF95BB
F3AC392526F01D4CD13E684C63A17CC4
D5F271642AD8389911381 7A61006413D
mod NIST P-384.

Maybe in some countries
the public is more demandin

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

BADA55ECD8BBEAD3ADD6C534F92197DE

B47FCEB9BE7E0E702A8D1DD56B5D0B0C

mod NIST P-256.

Similarly found $b = \texttt{0x}$

BADA55ECFD9CA54C0738B8A6FB8CF4CC

F84E916D83D6DA1B78B622351E11AB4E

mod NIST P-224; and $b = \texttt{0x}$

BADA55EC3BE2AD1F9EEEA5881ECF95BB

F3AC392526F01D4CD13E684C63A17CC4

D5F271642AD8389911381 7A61006413D

mod NIST P-384.

Maybe in some countries the public is more demanding.

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 D5F271642AD8389913817A61006413D mod NIST P-384.

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."

ed this computation on
of 41 GTX780 GPUs using
eccak. In 7 hours found
+twist-secure" $b = 0x$
CD8BBEAD3ADD6C534F92197DE
9BE7E0E702A8D1DD56B5D0B0C
ST P-256.
y found $b = 0x$
CFD9CA54C0738B8A6FB8CF4CC
D83D6DA1B78B622351E11AB4E
ST P-224; and $b = 0x$
C3BE2AD1F9EEEA5881ECF95BB
526F01D4CD13E684C63A17CC4
42AD83899113817A61006413D
ST P-384.

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
seedbytes = 20

p = 0xD7C134AA264
k = GF(p); R.<x> =

def secure(A,B):
  if k(B).is_squa
  n = EllipticCur
  return (n < p a
    and Integers(

def int2str(seed,
  return ''.join(

def str2int(seed)
  return Integer(

def update(seed):
  return int2str(

def fullhash(seed
  return str2int(

def real2str(seed
  return int2str(

nums = real2str(e
S = nums[2*seedby
while True:
  A = fullhash(S)
  if not (k(A)*x^
  S = update(S)
  B = fullhash(S)
  if not secure(A
  print 'p',hex(p
  print 'A',hex(A
  print 'B',hex(B
  break
```

mputation on
780 GPUs using
7 hours found
ure" $b =$ 0x
ADD6C534F92197DE
2A8D1DD56B5D0B0C

$=$ 0x
0738B8A6FB8CF4CC
78B622351E11AB4E
and $b =$ 0x
9EEEA5881ECF95BB
D13E684C63A17CC4
113817A61006413D

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.
seedbytes = 20

p = 0xD7C134AA264366862A18302575D1D787
k = GF(p); R.<x> = k[]

def secure(A,B):
  if k(B).is_square(): return False
  n = EllipticCurve([k(A),k(B)]).cardi
  return (n < p and n.is_prime()
    and Integers(n)(p).multiplicative_

def int2str(seed,bytes):
  return ''.join([chr((seed//256^i)%25

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

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

def fullhash(seed):
  return str2int(hash(seed) + hash(upd

def real2str(seed,bytes):
  return int2str(Integer(floor(RealFie

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 = upd
  S = update(S)
  B = fullhash(S)
  if not secure(A,B): S = update(S); c
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()
  break
```

on
using
und
0x
792197DE
3B5D0B0C
7B8CF4CC
1E11AB4E
0x
1ECF95BB
53A17CC4
1006413D

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
seedbytes = 20

p = 0xD7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C
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(

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)*2

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
```

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
```

n some countries
ic is more demanding.

ol standard (2005):
oice of the seeds
ich the [NIST] curve
ers have been derived is
ivated leaving an essential
he security analysis open.
ifiably pseudo-random.
ainpool] curves shall be
ed in a pseudo-random
using seeds that are
ed in a systematic and
ensive 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 caret
the curv
from the
Previous

Output

```
p D7C134AA2643
A 2B98B906DC24
B 68AEC4BFE84C
```

untries
demanding.

d (2005):
e seeds
IST] curve
been derived is
ving an essential
y analysis open.
eudo-random.
urves shall be
eudo-random
ls that are
tematic and
y."

```
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 imple
the curve-generati
from the Brainpoo
Previous slide: 224

Output of this pro

```
p  D7C134AA264366862A18302575D1D7
A  2B98B906DC245F2916C03A2F953EA9
B  68AEC4BFE84C659EBB8B81DC39355A
```

g.

e
ed is
sential
open.
dom.
be
om
nd

```
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 proced
from the Brainpool standard
Previous slide: 224-bit proce

Output of this procedure:

```
p D7C134AA264366862A18302575D1D787B09F075797DA89F
A 2B98B906DC245F2916C03A2F953EA9AE565C3253E8AEC4B
B 68AEC4BFE84C659EBB8B81DC39355A2EBFA3870D98976FA
```

```
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
```

```
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

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

p D7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF

A 68A5E62CA9CE6C1C299803A6C1530B514E182AD8B0042A59CAD29F43

B 2580F63CCFE44138870713B1A92369E33E2135D266DBB372386C400B

```
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
```

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.

```
 = hashlib.sha1(); h.update(seed); return h.digest()


366862A18302575D1D787B09F075797DA89F57EC8C0FF
= k[]


re(): return False
ve([k(A),k(B)]).cardinality()
nd n.is_prime()
n)(p).multiplicative_order() * 100 >= n-1)

bytes):
[chr((seed//256^i)%256) for i in reversed(range(bytes))])

:
seed.encode('hex'),16)


str2int(seed) + 1,len(seed))

):
hash(seed) + hash(update(seed))) % 2^223

,bytes):
Integer(floor(RealField(8*bytes+8)(seed)*256^bytes)),bytes)

xp(1)/16,7*seedbytes)
tes:3*seedbytes]


4+3).roots(): S = update(S); continue


,B): S = update(S); continue
).upper()
).upper()
).upper()
```

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
seedbytes = 20

p = 0xD7C134AA264
k = GF(p); R.<x> =

def secure(A,B):
  n = EllipticCur
  return (n < p a
    and Integers(

def int2str(seed,
  return ''.join(

def str2int(seed)
  return Integer(

def update(seed):
  return int2str(

def fullhash(seed
  return str2int(

def real2str(seed
  return int2str(

nums = real2str(e
S = nums[2*seedby
while True:
  A = fullhash(S)
  if not (k(A)*x^
  while True:
    S = update(S)
    B = fullhash(
    if not k(B).i
  if not secure(A
  print 'p',hex(p
  print 'A',hex(A
  print 'B',hex(B
  break
```

```
update(seed); return h.digest()

B09F075797DA89F57EC8C0FF



nality()

order() * 100 >= n-1)

6) for i in reversed(range(bytes))])

)

(seed))

ate(seed))) % 2^223

ld(8*bytes+8)(seed)*256^bytes)),bytes)



ate(S); continue

ontinue
```

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.
seedbytes = 20

p = 0xD7C134AA264366862A18302575D1D787
k = GF(p); R.<x> = k[]

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardi
  return (n < p and n.is_prime()
    and Integers(n)(p).multiplicative_

def int2str(seed,bytes):
  return ''.join([chr((seed//256^i)%25

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

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

def fullhash(seed):
  return str2int(hash(seed) + hash(upd

def real2str(seed,bytes):
  return int2str(Integer(floor(RealFie

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 = upd
  while True:
    S = update(S)
    B = fullhash(S)
    if not k(B).is_square(): break
  if not secure(A,B): S = update(S); c
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()
  break
```

```
h.digest()

C0FF

range(bytes))])

256^bytes)),bytes)
```

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
seedbytes = 20

p = 0xD7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C
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(

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)*2

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
```

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
```

fully implemented
e-generation procedure
e Brainpool standard.
s slide: 224-bit procedure.

of this procedure:

66862A18302575D1D787B09F075797DA89F57EC8C0FF
5F2916C03A2F953EA9AE565C3253E8AEC4BFE84C659E
659EBB8B81DC39355A2EBFA3870D98976FA2F17D2D8D

ndard 224-bit Brainpool
not the same curve:

66862A18302575D1D787B09F075797DA89F57EC8C0FF
6C1C299803A6C1530B514E182AD8B0042A59CAD29F43
4138870713B1A92369E33E2135D266DBB372386C400B

de: a procedure
es generate
dard 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 Brai
publicati
Did they

Brainpo
advertise
"compre
transpar
say the s

Can quie
to take t

emented
on procedure
l standard.
4-bit procedure.

cedure:

787B09F075797DA89F57EC8C0FF
9AE565C3253E8AEC4BFE84C659E
A2EBFA3870D98976FA2F17D2D8D

-bit Brainpool
same curve:

787B09F075797DA89F57EC8C0FF
3514E182AD8B0042A59CAD29F43
9E33E2135D266DBB372386C400B

edure
e
npool 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 che
publication? After
Did they know bef

Brainpool procedu
advertised as "syst
"comprehensive",
transparent", etc.
say the same for *b*

Can quietly manip
to take the weaker

ure
l.
edure.

57EC8C0FF
FE84C659E
2F17D2D8D

pool
e:

57EC8C0FF
9CAD29F43
2386C400B

e.

```
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 publicatio
Did they know before 2015?

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

Can quietly manipulate choi
to take the weaker procedur

```
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.

```
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.

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

```
 = hashlib.sha1(); h.update(seed); return h.digest()


366862A18302575D1D787B09F075797DA89F57EC8C0FF
= k[]


ve([k(A),k(B)]).cardinality()
nd n.is_prime()
n)(p).multiplicative_order() * 100 >= n-1)

bytes):
[chr((seed//256^i)%256) for i in reversed(range(bytes))])

:
seed.encode('hex'),16)


str2int(seed) + 1,len(seed))

):
hash(seed) + hash(update(seed))) % 2^223

,bytes):
Integer(floor(RealField(8*bytes+8)(seed)*256^bytes)),bytes)

xp(1)/16,7*seedbytes)
tes:3*seedbytes]


4+3).roots(): S = update(S); continue


S)
s_square(): break
,B): S = update(S); continue
).upper()
).upper()
).upper()
```

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 mad
using sta

To avoid
complica
hash out
from SH
maximu
Also upg

Brainpo
and arct
uses sin(
We also
pattern

```
update(seed); return h.digest()

B09F075797DA89F57EC8C0FF


nality()

order() * 100 >= n-1)

6) for i in reversed(range(bytes))])

)

(seed))

ate(seed))) % 2^223

ld(8*bytes+8)(seed)*256^bytes)),bytes)


ate(S); continue


ontinue
```

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 2
using standard NI

To avoid Brainpoo
complications of c
hash outputs: We
from SHA-1 to sta
maximum-security
Also upgraded to
maximum twist se

Brainpool uses exp
and $\arctan(1) = \pi$
uses $\sin(1)$, so we
We also used muc
pattern of searchi

```
h.digest()

COFF

range(bytes))])

256^bytes)),bytes)
```

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 cur using standard NIST P-224

To avoid Brainpool's complications of concatenat hash outputs: We upgraded from SHA-1 to state-of-the- maximum-security SHA3-51 Also upgraded to requiring maximum twist security.

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

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.

inpool check before
ion? After publication?
y know before 2015?

ol procedure is
ed as "systematic",
ehensive", "completely
ent", etc. Surely we can
same for *both* procedures.

etly manipulate choice
the weaker procedure.

ng Brainpool quote: "It
oned to provide additional
n 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

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

def secure(A,B):
  n = EllipticCur
  return (n.is_pr
    and Integers(
    and Integers(

def int2str(seed,
  return ''.join(

def str2int(seed)
  return Integer(

def complement(se
  return ''.join(

def real2str(seed
  return int2str(

sizeofint = 4
nums = real2str(c
for counter in xr
  S = int2str(cou
  T = complement(
  A = str2int(has
  B = str2int(has
  if secure(A,B):
    print 'p',hex
    print 'A',hex
    print 'B',hex
    break
```

ck before
publication?
fore 2015?

re is
tematic",
"completely
Surely we can
both procedures.

ulate choice
procedure.

ool quote: "It
ovide additional
r 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)]).cardi
  return (n.is_prime() and (2*p+2-n).i
    and Integers(n)(p).multiplicative_
    and Integers(2*p+2-n)(p).multiplic

def int2str(seed,bytes):
  return ''.join([chr((seed//256^i)%25

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

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

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

sizeofint = 4
nums = real2str(cos(1),seedbytes - siz
for counter in xrange(0,256^sizeofint)
  S = int2str(counter,sizeofint) + num
  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
```

on?

ely
e can
dures.

ce
e.

“It
itional

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 >

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

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

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
```

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
```

e a new 224-bit curve
andard NIST P-224 prime.

d Brainpool's
ations of concatenating
tputs: We upgraded
lA-1 to state-of-the-art
m-security SHA3-512.
graded to requiring
m twist security.

ol uses $\exp(1) = e$
an(1) $= \pi/4$, and MD5
(1), so we used cos(1).
used much simpler
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:

24-bit curve
ST P-224 prime.

ol's
oncatenating
upgraded
ate-of-the-art
SHA3-512.
requiring
curity.

$\rho(1) = e$
$\pi/4$, and MD5
used cos(1).
h simpler
ng 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: `7144BA12CE8A`

ve
prime.

ing

art
2.

MD5
1).

ds.

```
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: 7144BA12CE8A0C3BEFA053EDB

```
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...`

```
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...`

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).

```
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...`

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.