# Standardization for the black hat

Daniel J. Bernstein

University of Illinois at Chicago &
Technische Universiteit Eindhoven

---

(1) bada55.cr.yp.to "BADA55 Crypto" including "How to manipulate curve standards: a white paper for the black hat."

(2) projectbullrun.org including "Dual EC: a standardized back door."



Includes joint work with (in alphabetical order):

Tung Chou (1)
Chitchanok Chuengsatiansup (1)
Andreas Hülsing (1)
Eran Lambooij (1)
Tanja Lange (1) (2)
Ruben Niederhagen (1) (2)
Christine van Vredendaal (1)

Inspirational previous work: ANSI, ANSSI, Brainpool, IETF, ISO, NIST, OSCCA, SECG, and especially our buddies at NSA.

dization for the black hat

. Bernstein

ty of Illinois at Chicago &
che Universiteit Eindhoven

a55.cr.yp.to "BADA55
including "How to
ate curve standards: a
per for the black hat."

jectbullrun.org
g "Dual EC: a
lized back door."



Includes joint work with
(in alphabetical order):

Tung Chou (1)
Chitchanok Chuengsatiansup (1)
Andreas Hülsing (1)
Eran Lambooij (1)
Tanja Lange (1) (2)
Ruben Niederhagen (1) (2)
Christine van Vredendaal (1)

Inspirational previous work:
ANSI, ANSSI, Brainpool, IETF,
ISO, NIST, OSCCA, SECG, and
especially our buddies at NSA.



The DES

IBM: 12
IBM: 64
Final co

r the black hat

n

is at Chicago &
siteit Eindhoven

---

p.to "BADA55
"How to
standards: a
e black hat."

run.org
C: a
door."



Includes joint work with
(in alphabetical order):

Tung Chou (1)
Chitchanok Chuengsatiansup (1)
Andreas Hülsing (1)
Eran Lambooij (1)
Tanja Lange (1) (2)
Ruben Niederhagen (1) (2)
Christine van Vredendaal (1)

Inspirational previous work:
ANSI, ANSSI, Brainpool, IETF,
ISO, NIST, OSCCA, SECG, and
especially our buddies at NSA.



The DES key size

IBM: 128! NSA: 3
IBM: 64! NSA: 48
Final compromise:

k hat

ago &
hoven

DA55

a
t."



Includes joint work with
(in alphabetical order):

Tung Chou (1)
Chitchanok Chuengsatiansup (1)
Andreas Hülsing (1)
Eran Lambooij (1)
Tanja Lange (1) (2)
Ruben Niederhagen (1) (2)
Christine van Vredendaal (1)

Inspirational previous work:
ANSI, ANSSI, Brainpool, IETF,
ISO, NIST, OSCCA, SECG, and
especially our buddies at NSA.



The DES key size

IBM: 128! NSA: 32!
IBM: 64! NSA: 48!
Final compromise: 56.

Includes joint work with
(in alphabetical order):

Tung Chou (1)
Chitchanok Chuengsatiansup (1)
Andreas Hülsing (1)
Eran Lambooij (1)
Tanja Lange (1) (2)
Ruben Niederhagen (1) (2)
Christine van Vredendaal (1)

Inspirational previous work:
ANSI, ANSSI, Brainpool, IETF,
ISO, NIST, OSCCA, SECG, and
especially our buddies at NSA.



## The DES key size

IBM: 128! NSA: 32!
IBM: 64! NSA: 48!
Final compromise: 56.

Includes joint work with
(in alphabetical order):

Tung Chou (1)
Chitchanok Chuengsatiansup (1)
Andreas Hülsing (1)
Eran Lambooij (1)
Tanja Lange (1) (2)
Ruben Niederhagen (1) (2)
Christine van Vredendaal (1)

Inspirational previous work:
ANSI, ANSSI, Brainpool, IETF,
ISO, NIST, OSCCA, SECG, and
especially our buddies at NSA.



## The DES key size

IBM: 128! NSA: 32!
IBM: 64! NSA: 48!
Final compromise: 56.

Crypto community to NSA+NBS:
Your key size is too small.

Includes joint work with
(in alphabetical order):

Tung Chou (1)
Chitchanok Chuengsatiansup (1)
Andreas Hülsing (1)
Eran Lambooij (1)
Tanja Lange (1) (2)
Ruben Niederhagen (1) (2)
Christine van Vredendaal (1)

Inspirational previous work:
ANSI, ANSSI, Brainpool, IETF,
ISO, NIST, OSCCA, SECG, and
especially our buddies at NSA.



## The DES key size

IBM: 128! NSA: 32!
IBM: 64! NSA: 48!
Final compromise: 56.

Crypto community to NSA+NBS:
Your key size is too small.

NBS: Our key is big enough!
And we know how to use it!

Includes joint work with
(in alphabetical order):

Tung Chou (1)
Chitchanok Chuengsatiansup (1)
Andreas Hülsing (1)
Eran Lambooij (1)
Tanja Lange (1) (2)
Ruben Niederhagen (1) (2)
Christine van Vredendaal (1)

Inspirational previous work:
ANSI, ANSSI, Brainpool, IETF,
ISO, NIST, OSCCA, SECG, and
especially our buddies at NSA.



## The DES key size

IBM: 128! NSA: 32!
IBM: 64! NSA: 48!
Final compromise: 56.

Crypto community to NSA+NBS:
Your key size is too small.

NBS: Our key is big enough!
And we know how to use it!

NBS (now NIST) continues to
promote DES for two decades,
drastically increasing cost
of the inevitable upgrade.

joint work with
abetical order):

hou 1
hok Chuengsatiansup 1
Hülsing 1
mbooij 1
ange 1 2
Niederhagen 1 2
e van Vredendaal 1

onal previous work:
ANSSI, Brainpool, IETF,
ST, OSCCA, SECG, and
y our buddies at NSA.



The DES key size

IBM: 128! NSA: 32!
IBM: 64! NSA: 48!
Final compromise: 56.

Crypto community to NSA+NBS:
Your key size is too small.

NBS: Our key is big enough!
And we know how to use it!

NBS (now NIST) continues to
promote DES for two decades,
drastically increasing cost
of the inevitable upgrade.



Random

1992 Riv
given en
to hang
a standa

Standard

k with
der):

gsatiansup (1)
(1)
)
(2)
en (1) (2)
lendaal (1)

ous work:
inpool, IETF,
A, SECG, and
dies at NSA.



## The DES key size

IBM: 128! NSA: 32!
IBM: 64! NSA: 48!
Final compromise: 56.

Crypto community to NSA+NBS:
Your key size is too small.

NBS: Our key is big enough!
And we know how to use it!

NBS (now NIST) continues to
promote DES for two decades,
drastically increasing cost
of the inevitable upgrade.



## Random nonces in

1992 Rivest: “The
given enough rope
to hang himself—s
a standard should

Standardize anywa

o 1

TF,
and
A.



## The DES key size

IBM: 128! NSA: 32!
IBM: 64! NSA: 48!
Final compromise: 56.

Crypto community to NSA+NBS:
Your key size is too small.

NBS: Our key is big enough!
And we know how to use it!

NBS (now NIST) continues to promote DES for two decades, drastically increasing cost of the inevitable upgrade.



## Random nonces in DSA/EC

1992 Rivest: "The poor use
given enough rope with whic
to hang himself—something
a standard should not do."

Standardize anyway.

## The DES key size

IBM: 128! NSA: 32!
IBM: 64! NSA: 48!
Final compromise: 56.

Crypto community to NSA+NBS:
Your key size is too small.

NBS: Our key is big enough!
And we know how to use it!

NBS (now NIST) continues to promote DES for two decades, drastically increasing cost of the inevitable upgrade.



## Random nonces in DSA/ECDSA

1992 Rivest: "The poor user is given enough rope with which to hang himself—something a standard should not do."

Standardize anyway.

## The DES key size

IBM: 128! NSA: 32!
IBM: 64! NSA: 48!
Final compromise: 56.

Crypto community to NSA+NBS:
Your key size is too small.

NBS: Our key is big enough!
And we know how to use it!

NBS (now NIST) continues to promote DES for two decades, drastically increasing cost of the inevitable upgrade.



## Random nonces in DSA/ECDSA

1992 Rivest: "The poor user is given enough rope with which to hang himself—something a standard should not do."

Standardize anyway.

2010 Bushing–Marcan–Segher–Sven "PS3 epic fail": PS3 forgeries—Sony hung itself.

## The DES key size

IBM: 128! NSA: 32!
IBM: 64! NSA: 48!
Final compromise: 56.

Crypto community to NSA+NBS:
Your key size is too small.

NBS: Our key is big enough!
And we know how to use it!

NBS (now NIST) continues to promote DES for two decades, drastically increasing cost of the inevitable upgrade.



## Random nonces in DSA/ECDSA

1992 Rivest: "The poor user is given enough rope with which to hang himself—something a standard should not do."

Standardize anyway.

2010 Bushing–Marcan–Segher–Sven "PS3 epic fail": PS3 forgeries—Sony hung itself.

Add complicated *options* for deterministic nonces, while preserving old options.

S key size

8! NSA: 32!
! NSA: 48!
mpromise: 56.

community to NSA+NBS:
y size is too small.

ur key is big enough!
know how to use it!

ow NIST) continues to
DES for two decades,
ly increasing cost
nevitable upgrade.



Random nonces in DSA/ECDSA

1992 Rivest: "The poor user is given enough rope with which to hang himself—something a standard should not do."

Standardize anyway.

2010 Bushing–Marcan–Segher–Sven "PS3 epic fail": PS3 forgeries—Sony hung itself.

Add complicated *options* for deterministic nonces, while preserving old options.



Denial o

Suspecte
Bob are
"auditor
= "revie
exploitab
in crypto

Example
involved
around t
years of
How car
problem

32!
3!
56.

to NSA+NBS:
o small.

ig enough!
to use it!

continues to
two decades,
ng cost
pgrade.



## Random nonces in DSA/ECDSA

1992 Rivest: “The poor user is given enough rope with which to hang himself—something a standard should not do.”

Standardize anyway.

2010 Bushing–Marcan–Segher–Sven “PS3 epic fail”: PS3 forgeries—Sony hung itself.

Add complicated *options* for deterministic nonces, while preserving old options.



## Denial of service v

Suspected terrorist
Bob are aided and
“auditors” (= “cry
= “reviewers”) ch
exploitable security
in cryptographic sy

Example: SHA-3 c
involved 200 crypt
around the world a
years of sustained
How can we slip a
problem past all of

-NBS:

!

to

es,



## Random nonces in DSA/ECDSA

1992 Rivest: "The poor user is given enough rope with which to hang himself—something a standard should not do."

Standardize anyway.

2010 Bushing–Marcan–Segher–Sven "PS3 epic fail": PS3 forgeries—Sony hung itself.

Add complicated *options* for deterministic nonces, while preserving old options.



## Denial of service via flooding

Suspected terrorists Alice an Bob are aided and abetted b "auditors" (= "cryptanalysts = "reviewers") checking for exploitable security problems in cryptographic systems.

Example: SHA-3 competitio involved 200 cryptographers around the world and took years of sustained public effo How can we slip a security problem past all of them?

## Random nonces in DSA/ECDSA

1992 Rivest: “The poor user is given enough rope with which to hang himself—something a standard should not do.”

Standardize anyway.

2010 Bushing–Marcan–Segher–Sven “PS3 epic fail”: PS3 forgeries—Sony hung itself.

Add complicated *options* for deterministic nonces, while preserving old options.



## Denial of service via flooding

Suspected terrorists Alice and Bob are aided and abetted by “auditors” (= “cryptanalysts” = “reviewers”) checking for exploitable security problems in cryptographic systems.

Example: SHA-3 competition involved 200 cryptographers around the world and took years of sustained public effort. How can we slip a security problem past all of them?

## nonces in DSA/ECDSA

vest: "The poor user is
ough rope with which
himself—something
rd should not do."

dize anyway.

shing–Marcan–Segher–
S3 epic fail": PS3
—Sony hung itself.

nplicated *options*
rministic nonces,
eserving old options.



## Denial of service via flooding

Suspected terrorists Alice and Bob are aided and abetted by "auditors" (= "cryptanalysts" = "reviewers") checking for exploitable security problems in cryptographic systems.

Example: SHA-3 competition involved 200 cryptographers around the world and took years of sustained public effort. How can we slip a security problem past all of them?

During t
NIST als
FIPS 18
FIPS 19
SP 800-
SP 800-
SP 800-
SP 800-
SP 800-
SP 800-
SP 800-
SP 800-
SP 800-
and rela
such as

n DSA/ECDSA

e poor user is
e with which
something
not do."

ay.

rcan–Segher–
il": PS3
ung itself.

*options*
onces,
ld options.



## Denial of service via flooding

Suspected terrorists Alice and Bob are aided and abetted by "auditors" (= "cryptanalysts" = "reviewers") checking for exploitable security problems in cryptographic systems.

Example: SHA-3 competition involved 200 cryptographers around the world and took years of sustained public effort. How can we slip a security problem past all of them?



During the same p
NIST also publishe
FIPS 186-3 (signa
FIPS 198-1 (authe
SP 800-38E (disk
SP 800-38F (key w
SP 800-56C (key d
SP 800-57 (key m
SP 800-67 (block
SP 800-108 (key d
SP 800-131A (key
SP 800-133 (key g
SP 800-152 (key m
and related protoc
such as SP 800-81

DSA

r is

ch

er–

## Denial of service via flooding

Suspected terrorists Alice and Bob are aided and abetted by "auditors" (= "cryptanalysts" = "reviewers") checking for exploitable security problems in cryptographic systems.

Example: SHA-3 competition involved 200 cryptographers around the world and took years of sustained public effort.

How can we slip a security problem past all of them?



During the same period,
NIST also published
FIPS 186-3 (signatures),
FIPS 198-1 (authentication)
SP 800-38E (disk encryptio
SP 800-38F (key wrapping),
SP 800-56C (key derivation)
SP 800-57 (key managemen
SP 800-67 (block encryptio
SP 800-108 (key derivation)
SP 800-131A (key lengths),
SP 800-133 (key generation)
SP 800-152 (key manageme
and related protocol docume
such as SP 800-81r1.

## Denial of service via flooding

Suspected terrorists Alice and Bob are aided and abetted by "auditors" (= "cryptanalysts" = "reviewers") checking for exploitable security problems in cryptographic systems.

Example: SHA-3 competition involved 200 cryptographers around the world and took years of sustained public effort. How can we slip a security problem past all of them?



During the same period,
NIST also published
FIPS 186-3 (signatures),
FIPS 198-1 (authentication),
SP 800-38E (disk encryption),
SP 800-38F (key wrapping),
SP 800-56C (key derivation),
SP 800-57 (key management),
SP 800-67 (block encryption),
SP 800-108 (key derivation),
SP 800-131A (key lengths),
SP 800-133 (key generation),
SP 800-152 (key management),
and related protocol documents such as SP 800-81r1.

f service via flooding

ed terrorists Alice and
aided and abetted by
s" (= "cryptanalysts"
ewers") checking for
ble security problems
ographic systems.

e: SHA-3 competition
200 cryptographers
the world and took
sustained public effort.
n we slip a security
past all of them?





During the same period,
NIST also published
FIPS 186-3 (signatures),
FIPS 198-1 (authentication),
SP 800-38E (disk encryption),
SP 800-38F (key wrapping),
SP 800-56C (key derivation),
SP 800-57 (key management),
SP 800-67 (block encryption),
SP 800-108 (key derivation),
SP 800-131A (key lengths),
SP 800-133 (key generation),
SP 800-152 (key management),
and related protocol documents
such as SP 800-81r1.

Attentio
not enti

Auditors
security
just befo

ia flooding

ts Alice and
abetted by
yptanalysts"
ecking for
y problems
ystems.

competition
ographers
and took
public effort.
security
f them?



During the same period,
NIST also published
FIPS 186-3 (signatures),
FIPS 198-1 (authentication),
SP 800-38E (disk encryption),
SP 800-38F (key wrapping),
SP 800-56C (key derivation),
SP 800-57 (key management),
SP 800-67 (block encryption),
SP 800-108 (key derivation),
SP 800-131A (key lengths),
SP 800-133 (key generation),
SP 800-152 (key management),
and related protocol documents
such as SP 800-81r1.



Attention of audit
not entirely on SH

Auditors caught a
security flaw in EA
just before NIST s

g

nd

by

s”

s

on

ort.



During the same period,
NIST also published
FIPS 186-3 (signatures),
FIPS 198-1 (authentication),
SP 800-38E (disk encryption),
SP 800-38F (key wrapping),
SP 800-56C (key derivation),
SP 800-57 (key management),
SP 800-67 (block encryption),
SP 800-108 (key derivation),
SP 800-131A (key lengths),
SP 800-133 (key generation),
SP 800-152 (key management),
and related protocol documents
such as SP 800-81r1.



Attention of auditors was not entirely on SHA-3.

Auditors caught a severe security flaw in EAX Prime just before NIST standardiza

During the same period,
NIST also published
FIPS 186-3 (signatures),
FIPS 198-1 (authentication),
SP 800-38E (disk encryption),
SP 800-38F (key wrapping),
SP 800-56C (key derivation),
SP 800-57 (key management),
SP 800-67 (block encryption),
SP 800-108 (key derivation),
SP 800-131A (key lengths),
SP 800-133 (key generation),
SP 800-152 (key management),
and related protocol documents
such as SP 800-81r1.



Attention of auditors was not entirely on SHA-3.

Auditors caught a severe security flaw in EAX Prime just before NIST standardization.

During the same period,
NIST also published
FIPS 186-3 (signatures),
FIPS 198-1 (authentication),
SP 800-38E (disk encryption),
SP 800-38F (key wrapping),
SP 800-56C (key derivation),
SP 800-57 (key management),
SP 800-67 (block encryption),
SP 800-108 (key derivation),
SP 800-131A (key lengths),
SP 800-133 (key generation),
SP 800-152 (key management),
and related protocol documents
such as SP 800-81r1.



Attention of auditors was
not entirely on SHA-3.

Auditors caught a severe
security flaw in EAX Prime
just before NIST standardization.

Also a troublesome flaw in
the GCM security "proofs"
years *after* NIST standardization.

During the same period,
NIST also published
FIPS 186-3 (signatures),
FIPS 198-1 (authentication),
SP 800-38E (disk encryption),
SP 800-38F (key wrapping),
SP 800-56C (key derivation),
SP 800-57 (key management),
SP 800-67 (block encryption),
SP 800-108 (key derivation),
SP 800-131A (key lengths),
SP 800-133 (key generation),
SP 800-152 (key management),
and related protocol documents
such as SP 800-81r1.



Attention of auditors was not entirely on SHA-3.

Auditors caught a severe security flaw in EAX Prime just before NIST standardization.

Also a troublesome flaw in the GCM security "proofs" years *after* NIST standardization.

Why did this take years?
Scientific advances? No!
**We successfully denied service.**

During the same period,
NIST also published
FIPS 186-3 (signatures),
FIPS 198-1 (authentication),
SP 800-38E (disk encryption),
SP 800-38F (key wrapping),
SP 800-56C (key derivation),
SP 800-57 (key management),
SP 800-67 (block encryption),
SP 800-108 (key derivation),
SP 800-131A (key lengths),
SP 800-133 (key generation),
SP 800-152 (key management),
and related protocol documents
such as SP 800-81r1.



Attention of auditors was not entirely on SHA-3.

Auditors caught a severe security flaw in EAX Prime just before NIST standardization.

Also a troublesome flaw in the GCM security "proofs" years *after* NIST standardization.

Why did this take years? Scientific advances? No! **We successfully denied service.**

And NIST is just the tip of the crypto standardization iceberg.

the same period,
so published
6-3 (signatures),
8-1 (authentication),
38E (disk encryption),
38F (key wrapping),
56C (key derivation),
57 (key management),
67 (block encryption),
108 (key derivation),
131A (key lengths),
133 (key generation),
152 (key management),
ted protocol documents
SP 800-81r1.





Attention of auditors was not entirely on SHA-3.

Auditors caught a severe security flaw in EAX Prime just before NIST standardization.

Also a troublesome flaw in the GCM security "proofs" years *after* NIST standardization.

Why did this take years? Scientific advances? No! **We successfully denied service.**

And NIST is just the tip of the crypto standardization iceberg.

Flooding

If we we
would te
ciphers/

period,
ed
tures),
entication),
encryption),
wrapping),
derivation),
anagement),
encryption),
derivation),
lengths),
generation),
management),
col documents
.r1.



Attention of auditors was not entirely on SHA-3.

Auditors caught a severe security flaw in EAX Prime just before NIST standardization.

Also a troublesome flaw in the GCM security "proofs" years *after* NIST standardization.

Why did this take years? Scientific advances? No! **We successfully denied service.**

And NIST is just the tip of the crypto standardization iceberg.



Flooding via disho

If we were honest would tell Alice+B ciphers/hashes as

,
n),

),
t),
n),
,

),
nt),
ents



Attention of auditors was not entirely on SHA-3.

Auditors caught a severe security flaw in EAX Prime just before NIST standardization.

Also a troublesome flaw in the GCM security “proofs” years *after* NIST standardization.

Why did this take years?
Scientific advances? No!
**We successfully denied service.**

And NIST is just the tip of the crypto standardization iceberg.



Flooding via dishonesty

If we were honest then we would tell Alice+Bob to reu
ciphers/hashes as PRNGs.

Attention of auditors was not entirely on SHA-3.

Auditors caught a severe security flaw in EAX Prime just before NIST standardization.

Also a troublesome flaw in the GCM security “proofs” years *after* NIST standardization.

Why did this take years?
Scientific advances? No!
**We successfully denied service.**

And NIST is just the tip of the crypto standardization iceberg.



Flooding via dishonesty

If we were honest then we would tell Alice+Bob to reuse ciphers/hashes as PRNGs.

Attention of auditors was not entirely on SHA-3.

Auditors caught a severe security flaw in EAX Prime just before NIST standardization.

Also a troublesome flaw in the GCM security "proofs" years *after* NIST standardization.

Why did this take years? Scientific advances? No! **We successfully denied service.**

And NIST is just the tip of the crypto standardization iceberg.



## Flooding via dishonesty

If we were honest then we would tell Alice+Bob to reuse ciphers/hashes as PRNGs.

But why should we be honest? Let's build PRNGs from scratch!

Attention of auditors was not entirely on SHA-3.

Auditors caught a severe security flaw in EAX Prime just before NIST standardization.

Also a troublesome flaw in the GCM security "proofs" years *after* NIST standardization.

Why did this take years? Scientific advances? No! **We successfully denied service.**

And NIST is just the tip of the crypto standardization iceberg.



## Flooding via dishonesty

If we were honest then we would tell Alice+Bob to reuse ciphers/hashes as PRNGs.

But why should we be honest? Let's build PRNGs from scratch!

2004: Number-theoretic RNGs provide "increased assurance."

2006: Dual EC "is the only DRBG mechanism in this Recommendation whose security is related to a hard problem in number theory."

n of auditors was
rely on SHA-3.

caught a severe
flaw in EAX Prime
ore NIST standardization.

roublesome flaw in
M security "proofs"
*ter* NIST standardization.

this take years?
c advances? No!
**cessfully denied service.**

T is just the tip of the
tandardization iceberg.



Flooding via dishonesty

If we were honest then we would tell Alice+Bob to reuse ciphers/hashes as PRNGs.

But why should we be honest? Let's build PRNGs from scratch!

2004: Number-theoretic RNGs provide "increased assurance."

2006: Dual EC
"is the only DRBG mechanism in this Recommendation whose security is related to a hard problem in number theory."

Denial o

2006 Gj
2006 Sc
Dual EC
definitio

ors was
A-3.

severe
AX Prime
standardization.

e flaw in
"proofs"
standardization.

years?
s? No!
**denied service.**

the tip of the
tion iceberg.



## Flooding via dishonesty

If we were honest then we would tell Alice+Bob to reuse ciphers/hashes as PRNGs.

But why should we be honest? Let's build PRNGs from scratch!

2004: Number-theoretic RNGs provide "increased assurance."

2006: Dual EC
"is the only DRBG mechanism in this Recommendation whose security is related to a hard problem in number theory."



## Denial of service v

2006 Gjøsteen, inc
2006 Schoenmake
Dual EC flunks we
definition of PRN

ation.

ation.

**rvice.**

the

rg.



## Flooding via dishonesty

If we were honest then we would tell Alice+Bob to reuse ciphers/hashes as PRNGs.

But why should we be honest? Let's build PRNGs from scratch!

2004: Number-theoretic RNGs provide "increased assurance."

2006: Dual EC
"is the only DRBG mechanism in this Recommendation whose security is related to a hard problem in number theory."



## Denial of service via hoops

2006 Gjøsteen, independentl
2006 Schoenmakers–Sidoren
Dual EC flunks well-establisl
definition of PRNG security.

## Flooding via dishonesty

If we were honest then we would tell Alice+Bob to reuse ciphers/hashes as PRNGs.

But why should we be honest? Let's build PRNGs from scratch!

2004: Number-theoretic RNGs provide "increased assurance."

2006: Dual EC
"is the only DRBG mechanism in this Recommendation whose security is related to a hard problem in number theory."



## Denial of service via hoops

2006 Gjøsteen, independently
2006 Schoenmakers–Sidorenko:
Dual EC flunks well-established definition of PRNG security.

## Flooding via dishonesty

If we were honest then we would tell Alice+Bob to reuse ciphers/hashes as PRNGs.

But why should we be honest? Let's build PRNGs from scratch!

2004: Number-theoretic RNGs provide "increased assurance."

2006: Dual EC
"is the only DRBG mechanism in this Recommendation whose security is related to a hard problem in number theory."



## Denial of service via hoops

2006 Gjøsteen, independently
2006 Schoenmakers–Sidorenko:
Dual EC flunks well-established definition of PRNG security.

Are *all* applications broken?
Obviously not! Standardize!

## Flooding via dishonesty

If we were honest then we would tell Alice+Bob to reuse ciphers/hashes as PRNGs.

But why should we be honest? Let's build PRNGs from scratch!

2004: Number-theoretic RNGs provide "increased assurance."

2006: Dual EC
"is the only DRBG mechanism in this Recommendation whose security is related to a hard problem in number theory."



## Denial of service via hoops

2006 Gjøsteen, independently
2006 Schoenmakers–Sidorenko: Dual EC flunks well-established definition of PRNG security.

Are *all* applications broken? Obviously not! Standardize!

2007 Shumow–Ferguson: Dual EC has a back door. Would have been easy to build $Q$ with the key.

2007 Schneier: Never use Dual EC. "Both NIST and the NSA have some explaining to do."

g via dishonesty

re honest then we
ell Alice+Bob to reuse
hashes as PRNGs.

should we be honest?
ild PRNGs from scratch!

umber-theoretic RNGs
"increased assurance."

ual EC
only DRBG mechanism
Recommendation
ecurity is related to a
blem in number theory."



Denial of service via hoops

2006 Gjøsteen, independently
2006 Schoenmakers–Sidorenko:
Dual EC flunks well-established
definition of PRNG security.

Are *all* applications broken?
Obviously not! Standardize!

2007 Shumow–Ferguson: Dual
EC has a back door. Would have
been easy to build $Q$ with the key.

2007 Schneier: Never use Dual
EC. "Both NIST and the NSA
have some explaining to do."

Did Shu
show us

Maintai
standard

2008.07-
73 valida
for Dual

onesty

then we
Bob to reuse
PRNGs.

e be honest?
s from scratch!

eoretic RNGs
assurance."

G mechanism
dation
related to a
umber theory."



Denial of service via hoops

2006 Gjøsteen, independently
2006 Schoenmakers–Sidorenko:
Dual EC flunks well-established
definition of PRNG security.

Are *all* applications broken?
Obviously not! Standardize!

2007 Shumow–Ferguson: Dual
EC has a back door. Would have
been easy to build $Q$ with the key.

2007 Schneier: Never use Dual
EC. "Both NIST and the NSA
have some explaining to do."

Did Shumow and
show us the key?

Maintain and pron
standard. Pay peo

2008.07–2014.03:
73 validation certi
for Dual EC imple

se

st?

atch!

Gs

e.”

sm

a

ory.”





Denial of service via hoops

2006 Gjøsteen, independently
2006 Schoenmakers–Sidorenko:
Dual EC flunks well-established
definition of PRNG security.

Are *all* applications broken?
Obviously not! Standardize!

2007 Shumow–Ferguson: Dual
EC has a back door. Would have
been easy to build $Q$ with the key.

2007 Schneier: Never use Dual
EC. “Both NIST and the NSA
have some explaining to do.”

Did Shumow and Ferguson
show us the key? No!

Maintain and promote Dual
standard. Pay people to use

2008.07–2014.03: NIST issu
73 validation certificates
for Dual EC implementation

## Denial of service via hoops

2006 Gjøsteen, independently
2006 Schoenmakers–Sidorenko:
Dual EC flunks well-established definition of PRNG security.

Are *all* applications broken?
Obviously not! Standardize!

2007 Shumow–Ferguson: Dual EC has a back door. Would have been easy to build $Q$ with the key.

2007 Schneier: Never use Dual EC. "Both NIST and the NSA have some explaining to do."



Did Shumow and Ferguson show us the key? No!

Maintain and promote Dual EC standard. Pay people to use it.

2008.07–2014.03: NIST issues 73 validation certificates for Dual EC implementations.

## Denial of service via hoops

2006 Gjøsteen, independently
2006 Schoenmakers–Sidorenko:
Dual EC flunks well-established definition of PRNG security.

Are *all* applications broken?
Obviously not! Standardize!

2007 Shumow–Ferguson: Dual EC has a back door. Would have been easy to build $Q$ with the key.

2007 Schneier: Never use Dual EC. "Both NIST and the NSA have some explaining to do."



Did Shumow and Ferguson show us the key? No!

Maintain and promote Dual EC standard. Pay people to use it.

2008.07–2014.03: NIST issues 73 validation certificates for Dual EC implementations.

Even after being caught, continue to burn auditors' time by demanding that they jump higher.

NSA's Dickie George, 2014: Gee, Dual EC is really hard to exploit!

f service via hoops

østeen, independently
hoenmakers–Sidorenko:
 flunks well-established
n of PRNG security.

applications broken?
ly not! Standardize!

umow–Ferguson: Dual
a back door. Would have
sy to build $Q$ with the key.

hneier: Never use Dual
oth NIST and the NSA
me explaining to do.”



Did Shumow and Ferguson show us the key? No!

Maintain and promote Dual EC standard. Pay people to use it.

2008.07–2014.03: NIST issues 73 validation certificates for Dual EC implementations.

Even after being caught, continue to burn auditors’ time by demanding that they jump higher.

NSA’s Dickie George, 2014: Gee, Dual EC is really hard to exploit!

System

Traditio
Auditor
an RNG

Auditor’
random
Bob are

via hoops

dependently
rs–Sidorenko:
ell-established
G security.

ns broken?
andardize!

rguson: Dual
or. Would have
$Q$ with the key.

ever use Dual
and the NSA
ing to do."



Did Shumow and Ferguson show us the key? No!

Maintain and promote Dual EC standard. Pay people to use it.

2008.07–2014.03: NIST issues 73 validation certificates for Dual EC implementations.

Even after being caught, continue to burn auditors' time by demanding that they jump higher.

NSA's Dickie George, 2014: Gee, Dual EC is really hard to exploit!

System vs. ecosyst

Traditional RNG a
Auditor looks at o
an RNG. Tries to

Auditor's starting
random numbers f
Bob are created by

ly
ko:
hed

ual
have
he key.

ual
SA
”



Did Shumow and Ferguson show us the key? No!

Maintain and promote Dual EC standard. Pay people to use it.

2008.07–2014.03: NIST issues 73 validation certificates for Dual EC implementations.

Even after being caught, continue to burn auditors’ time by demanding that they jump higher.

NSA’s Dickie George, 2014: Gee, Dual EC is really hard to exploit!



System vs. ecosystem

Traditional RNG auditing:

Auditor looks at one system
an RNG. Tries to find weakn

Auditor’s starting assumptio
random numbers for Alice a
Bob are created by an RNG.

Did Shumow and Ferguson show us the key? No!

Maintain and promote Dual EC standard. Pay people to use it.

2008.07–2014.03: NIST issues 73 validation certificates for Dual EC implementations.

Even after being caught, continue to burn auditors' time by demanding that they jump higher.

NSA's Dickie George, 2014: Gee, Dual EC is really hard to exploit!



## System vs. ecosystem

Traditional RNG auditing:
Auditor looks at one system, an RNG. Tries to find weakness.

Auditor's starting assumption: random numbers for Alice and Bob are created by an RNG.

Did Shumow and Ferguson show us the key? No!

Maintain and promote Dual EC standard. Pay people to use it.

2008.07–2014.03: NIST issues 73 validation certificates for Dual EC implementations.

Even after being caught, continue to burn auditors' time by demanding that they jump higher.

NSA's Dickie George, 2014: Gee, Dual EC is really hard to exploit!



System vs. ecosystem

Traditional RNG auditing: Auditor looks at one system, an RNG. Tries to find weakness.

Auditor's starting assumption: random numbers for Alice and Bob are created by an RNG.

Reality: random numbers are created by a much more complicated ecosystem that designs, evaluates, standardizes, selects, implements, and deploys RNGs. (Same for other crypto.)

mow and Ferguson
the key? No!

n and promote Dual EC
l. Pay people to use it.

–2014.03: NIST issues
ation certificates
EC implementations.

ter being caught,
e to burn auditors' time by
ing that they jump higher.

Dickie George, 2014: Gee,
C is really hard to exploit!



System vs. ecosystem

Traditional RNG auditing:
Auditor looks at one system,
an RNG. Tries to find weakness.

Auditor's starting assumption:
random numbers for Alice and
Bob are created by an RNG.

Reality: random numbers
are created by a much more
complicated ecosystem that
designs, evaluates, standardizes,
selects, implements, and deploys
RNGs. (Same for other crypto.)

**This is a**
**perspec**
defendin

The eco
weaknes
inside ar

e.g. Easi

Ferguson
No!

mote Dual EC
ple to use it.

NIST issues
ficates
mentations.

aught,
auditors' time by
ey jump higher.

rge, 2014: Gee,
hard to exploit!



System vs. ecosystem

Traditional RNG auditing:
Auditor looks at one system,
an RNG. Tries to find weakness.

Auditor's starting assumption:
random numbers for Alice and
Bob are created by an RNG.

Reality: random numbers
are created by a much more
complicated ecosystem that
designs, evaluates, standardizes,
selects, implements, and deploys
RNGs. (Same for other crypto.)

**This is a critical**
**perspective.** Aud
defending the wro

The ecosystem ha
weaknesses that a
inside any particul

e.g. Easily take co

EC
it.

ues

s.

me by
higher.

Gee,
ploit!



System vs. ecosystem

Traditional RNG auditing: Auditor looks at one system, an RNG. Tries to find weakness.

Auditor's starting assumption: random numbers for Alice and Bob are created by an RNG.

Reality: random numbers are created by a much more complicated ecosystem that designs, evaluates, standardizes, selects, implements, and deploys RNGs. (Same for other crypto.)

**This is a critical change in perspective.** Auditor is stu
defending the wrong targets

The ecosystem has many
weaknesses that are not visi
inside any particular system.

e.g. Easily take control of IS

## System vs. ecosystem

Traditional RNG auditing:
Auditor looks at one system,
an RNG. Tries to find weakness.

Auditor's starting assumption:
random numbers for Alice and
Bob are created by an RNG.

Reality: random numbers
are created by a much more
complicated ecosystem that
designs, evaluates, standardizes,
selects, implements, and deploys
RNGs. (Same for other crypto.)



**This is a critical change in perspective.** Auditor is stuck
defending the wrong targets!

The ecosystem has many
weaknesses that are not visible
inside any particular system.

e.g. Easily take control of ISO.

## System vs. ecosystem

Traditional RNG auditing: Auditor looks at one system, an RNG. Tries to find weakness.

Auditor's starting assumption: random numbers for Alice and Bob are created by an RNG.

Reality: random numbers are created by a much more complicated ecosystem that designs, evaluates, standardizes, selects, implements, and deploys RNGs. (Same for other crypto.)



**This is a critical change in perspective.** Auditor is stuck defending the wrong targets!

The ecosystem has many weaknesses that are not visible inside any particular system.

e.g. Easily take control of ISO.

e.g. Propose 20 weak standards. Some will survive auditing. Then manipulate selection.

## System vs. ecosystem

Traditional RNG auditing:
Auditor looks at one system,
an RNG. Tries to find weakness.

Auditor's starting assumption:
random numbers for Alice and
Bob are created by an RNG.

Reality: random numbers
are created by a much more
complicated ecosystem that
designs, evaluates, standardizes,
selects, implements, and deploys
RNGs. (Same for other crypto.)



**This is a critical change in perspective.** Auditor is stuck defending the wrong targets!

The ecosystem has many weaknesses that are not visible inside any particular system.

e.g. Easily take control of ISO.

e.g. Propose 20 weak standards.
Some will survive auditing.
Then manipulate selection.

Deter publication of weaknesses:
"This attack is trivial. Reject."

vs. ecosystem

nal RNG auditing:
looks at one system,
. Tries to find weakness.

s starting assumption:
numbers for Alice and
created by an RNG.

random numbers
ted by a much more
ated ecosystem that
evaluates, standardizes,
implements, and deploys
(Same for other crypto.)



**This is a critical change in perspective.** Auditor is stuck defending the wrong targets!

The ecosystem has many weaknesses that are not visible inside any particular system.

e.g. Easily take control of ISO.

e.g. Propose 20 weak standards. Some will survive auditing. Then manipulate selection.

Deter publication of weaknesses: "This attack is trivial. Reject."

Textboo
using sta
on a sta

tem

uditing:
ne system,
find weakness.

assumption:
for Alice and
y an RNG.

umbers
much more
stem that
standardizes,
ts, and deploys
other crypto.)



**This is a critical change in perspective.** Auditor is stuck defending the wrong targets!

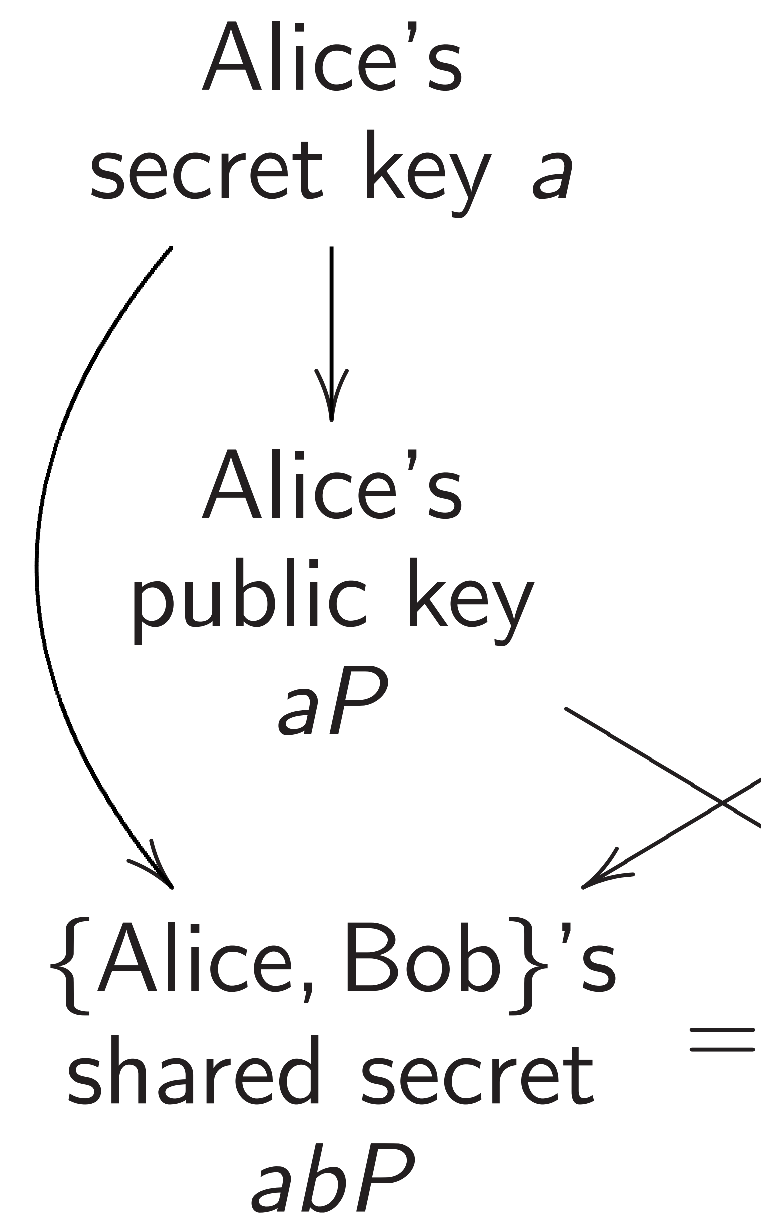
The ecosystem has many weaknesses that are not visible inside any particular system.

e.g. Easily take control of ISO.

e.g. Propose 20 weak standards. Some will survive auditing. Then manipulate selection.

Deter publication of weaknesses: “This attack is trivial. Reject.”

Textbook key exch
using standard poi
on a standard ellip



,
ness.
n:
nd
.
izes,
ploys
pto.)



**This is a critical change in perspective.** Auditor is stuck defending the wrong targets!

The ecosystem has many weaknesses that are not visible inside any particular system.

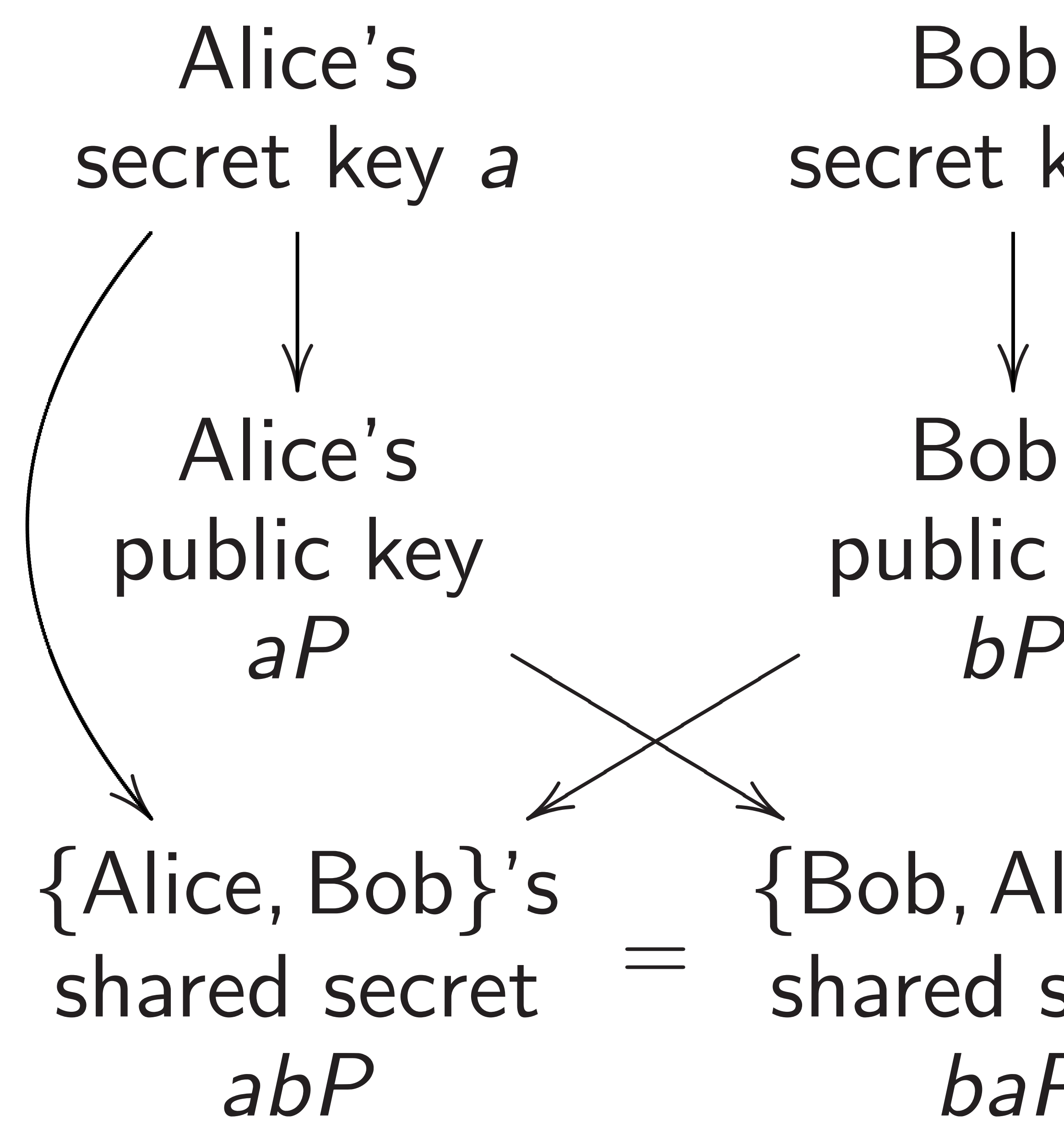
e.g. Easily take control of ISO.

e.g. Propose 20 weak standards. Some will survive auditing. Then manipulate selection.

Deter publication of weaknesses: "This attack is trivial. Reject."



Textbook key exchange using standard point $P$ on a standard elliptic curve



**This is a critical change in perspective.** Auditor is stuck defending the wrong targets!

The ecosystem has many weaknesses that are not visible inside any particular system.

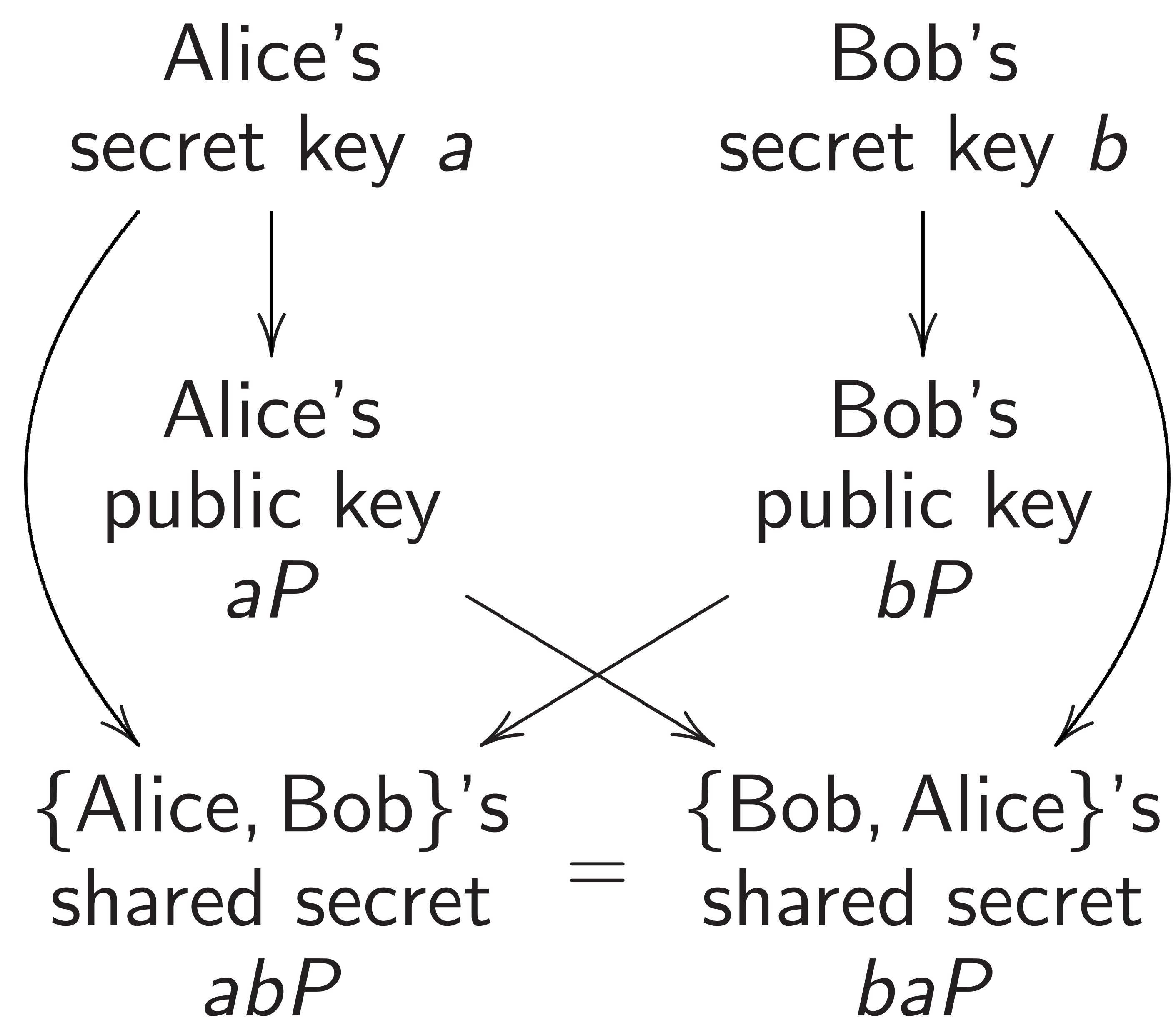
e.g. Easily take control of ISO.

e.g. Propose 20 weak standards. Some will survive auditing. Then manipulate selection.

Deter publication of weaknesses: "This attack is trivial. Reject."



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



**This is a critical change in perspective.** Auditor is stuck defending the wrong targets!

The ecosystem has many weaknesses that are not visible inside any particular system.

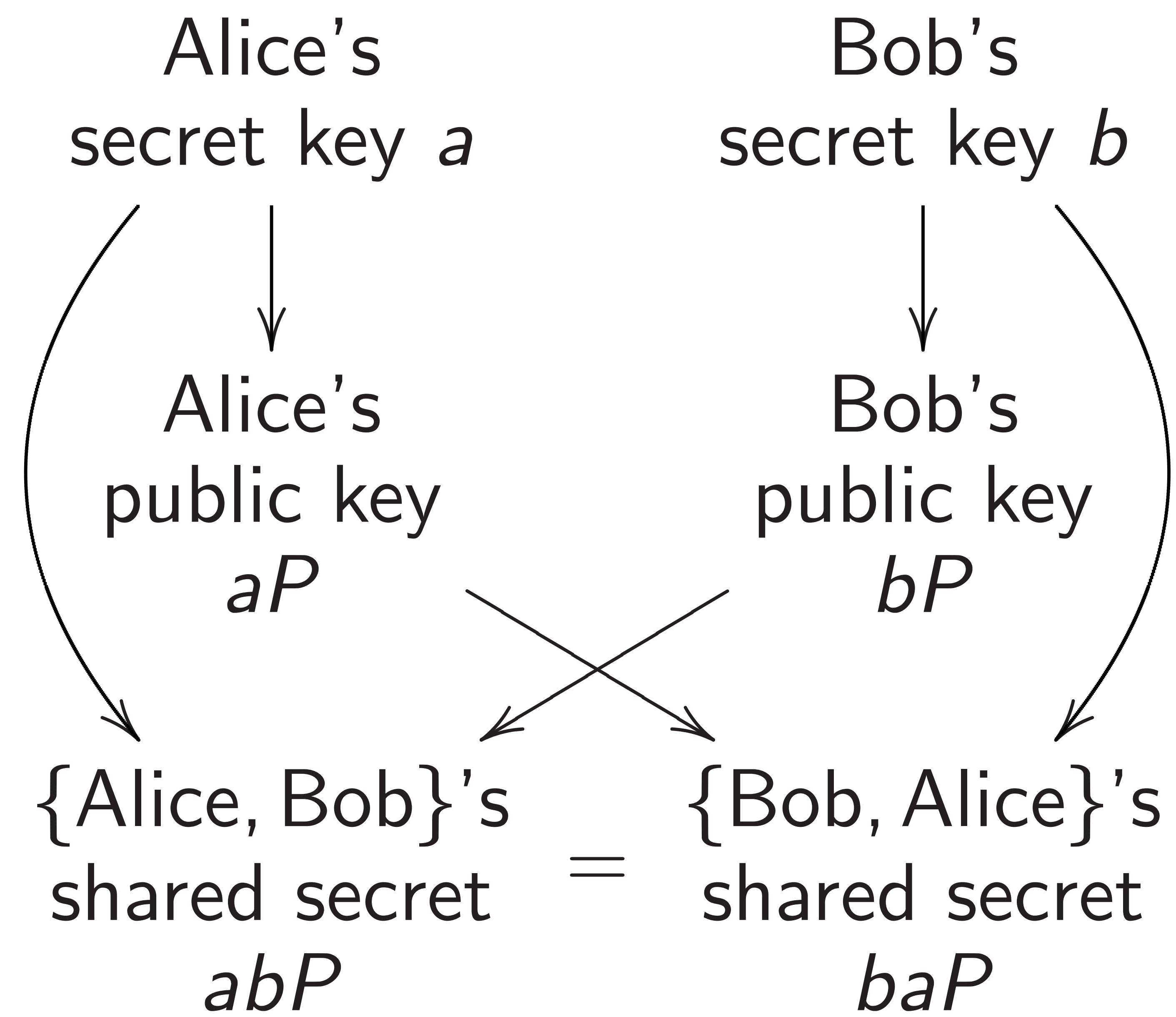
e.g. Easily take control of ISO.

e.g. Propose 20 weak standards. Some will survive auditing. Then manipulate selection.

Deter publication of weaknesses: "This attack is trivial. Reject."



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




Security depends on choice of $E$.

**a critical change in**
**tive.** Auditor is stuck
g the wrong targets!

system has many
ses that are not visible
ny particular system.

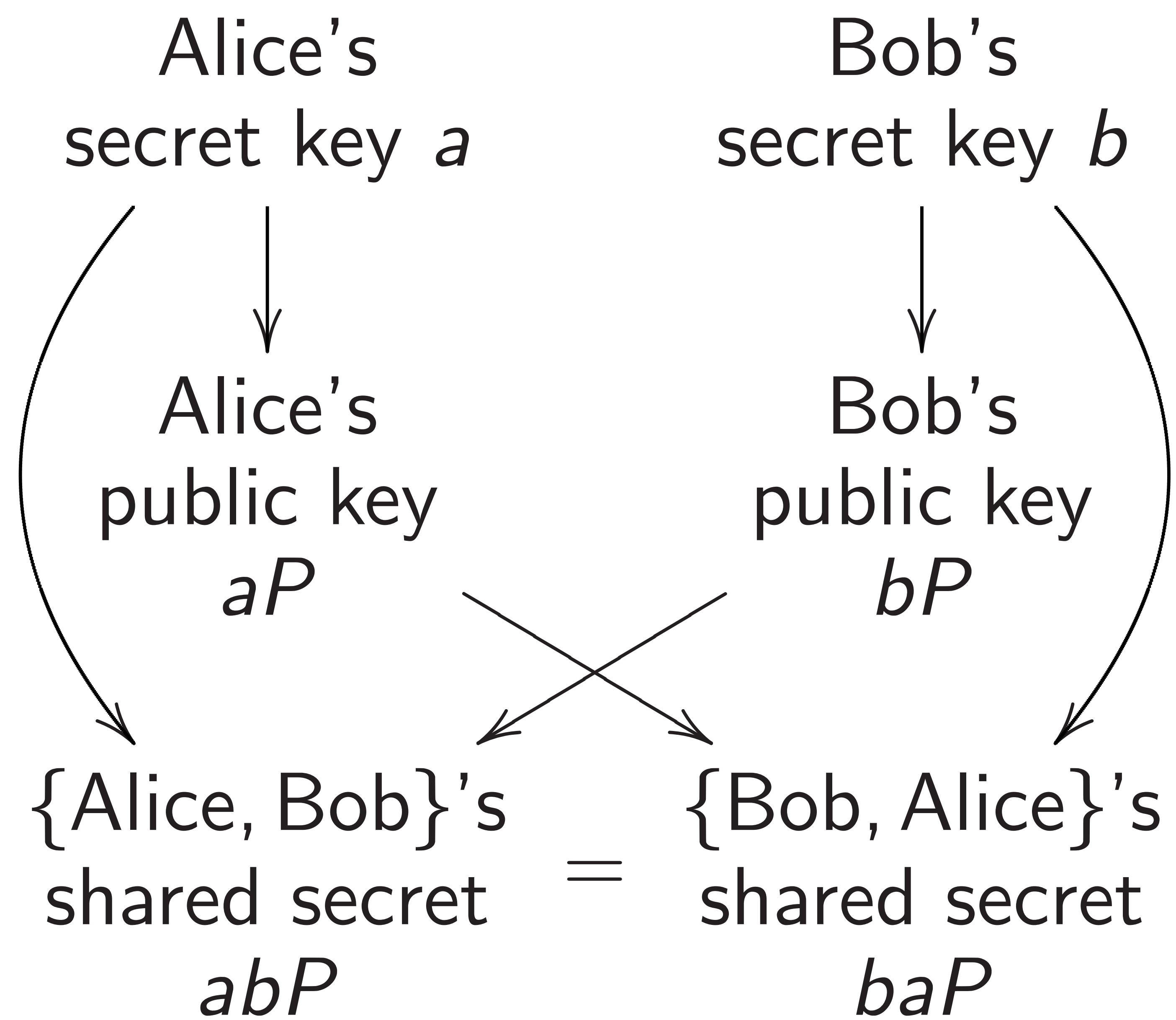
ily take control of ISO.

pose 20 weak standards.
ill survive auditing.
anipulate selection.

ublication of weaknesses:
tack is trivial. Reject."



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




Security depends on choice of $E$.

C


**This is**

**change in**
litor is stuck
ng targets!

s many
re not visible
ar system.

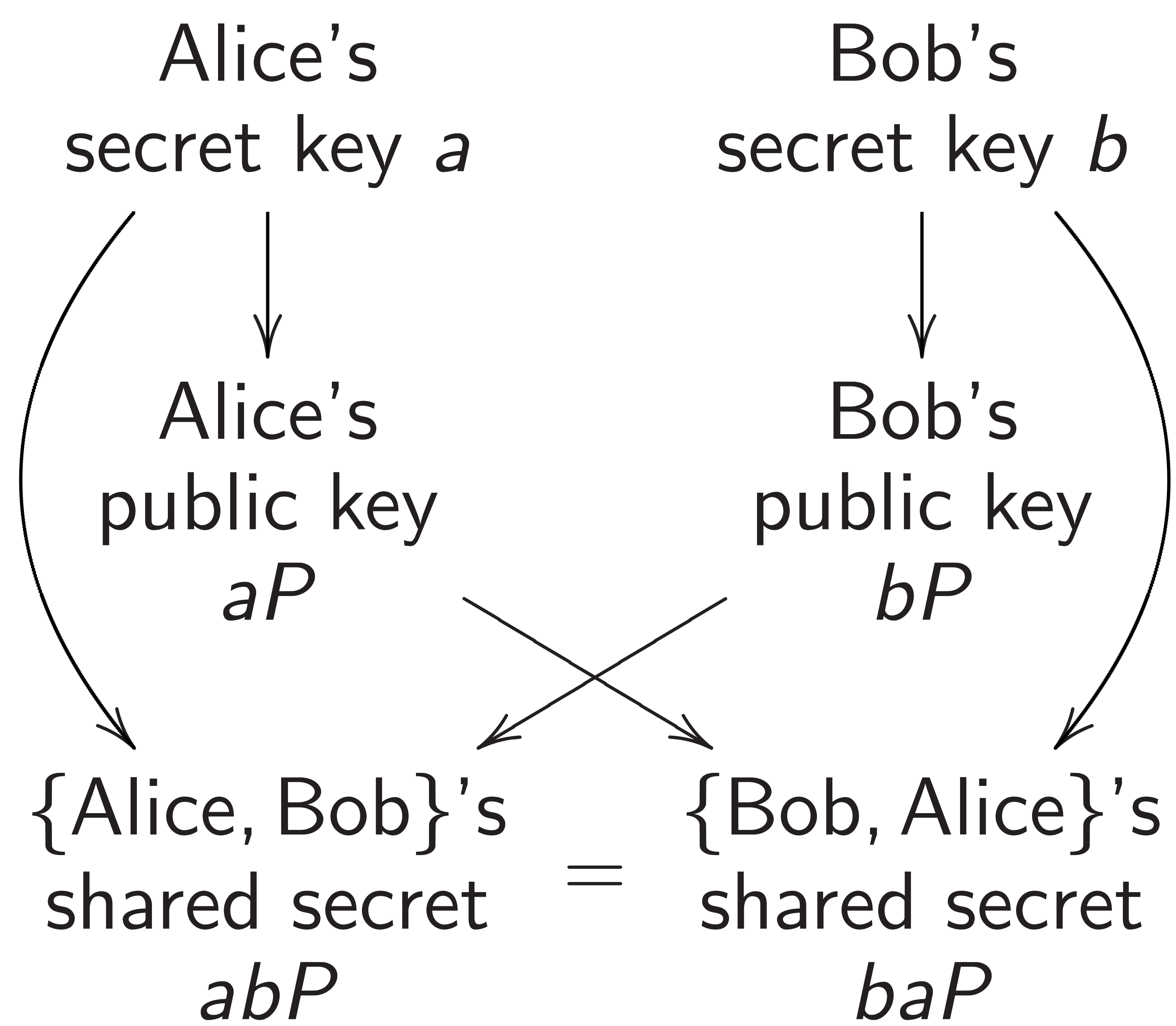
ntrol of ISO.

eak standards.
auditing.
selection.

of weaknesses:
vial. Reject."

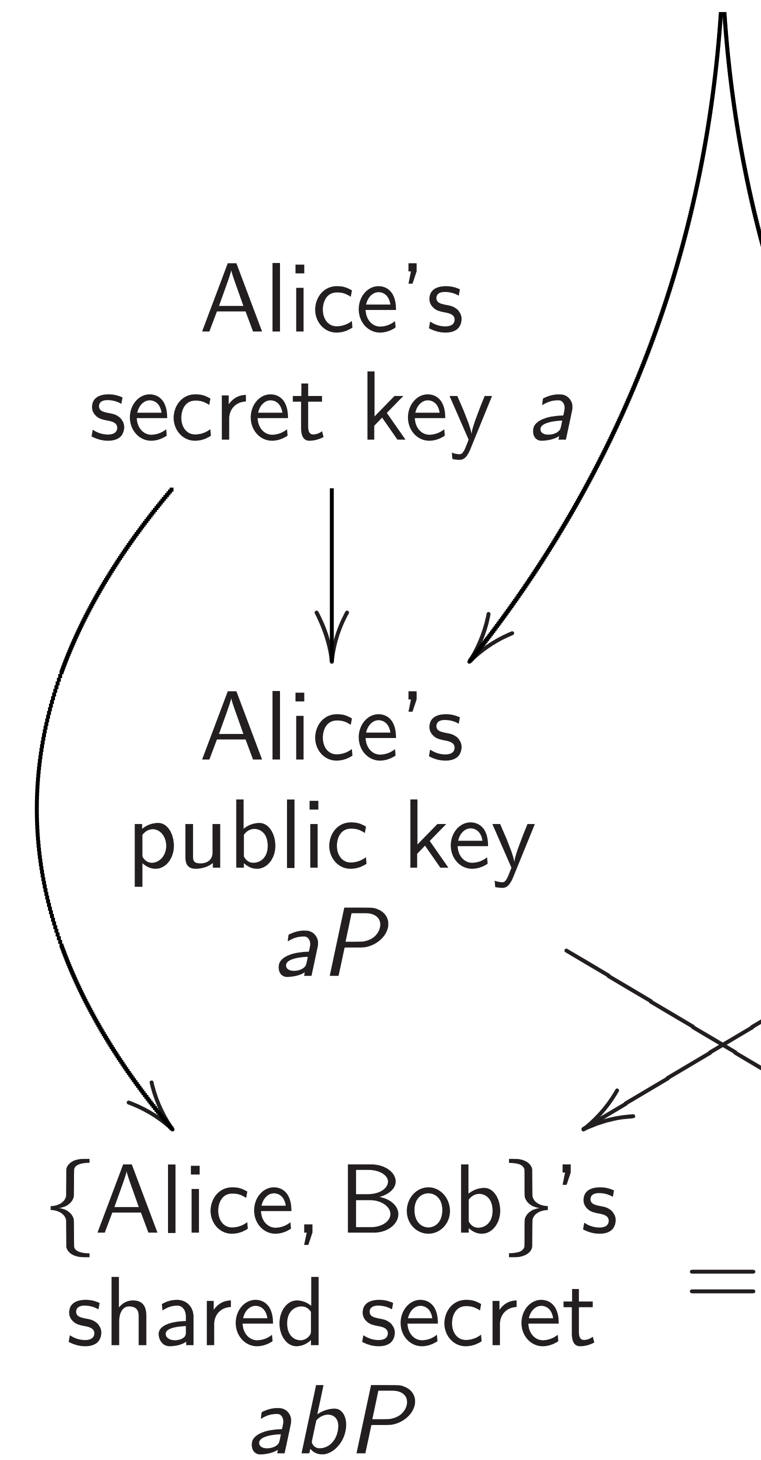


Textbook key exchange
using standard point $P$
on a standard elliptic curve $E$:




Security depends on choice of $E$.

Our partne
choice o




**This is not the sa**

n

ck

!

ble

SO.

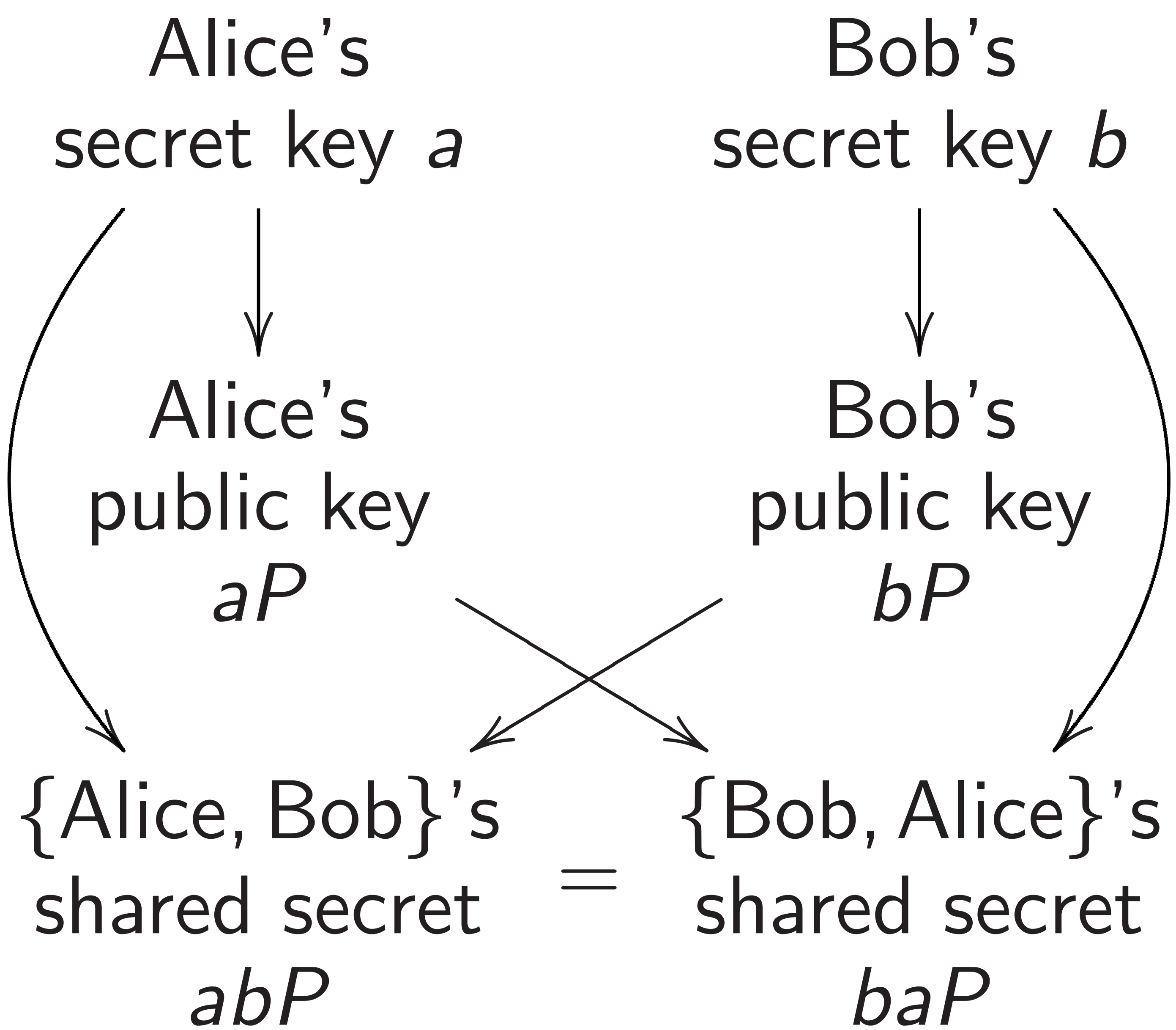
ards.

sses:

ct."

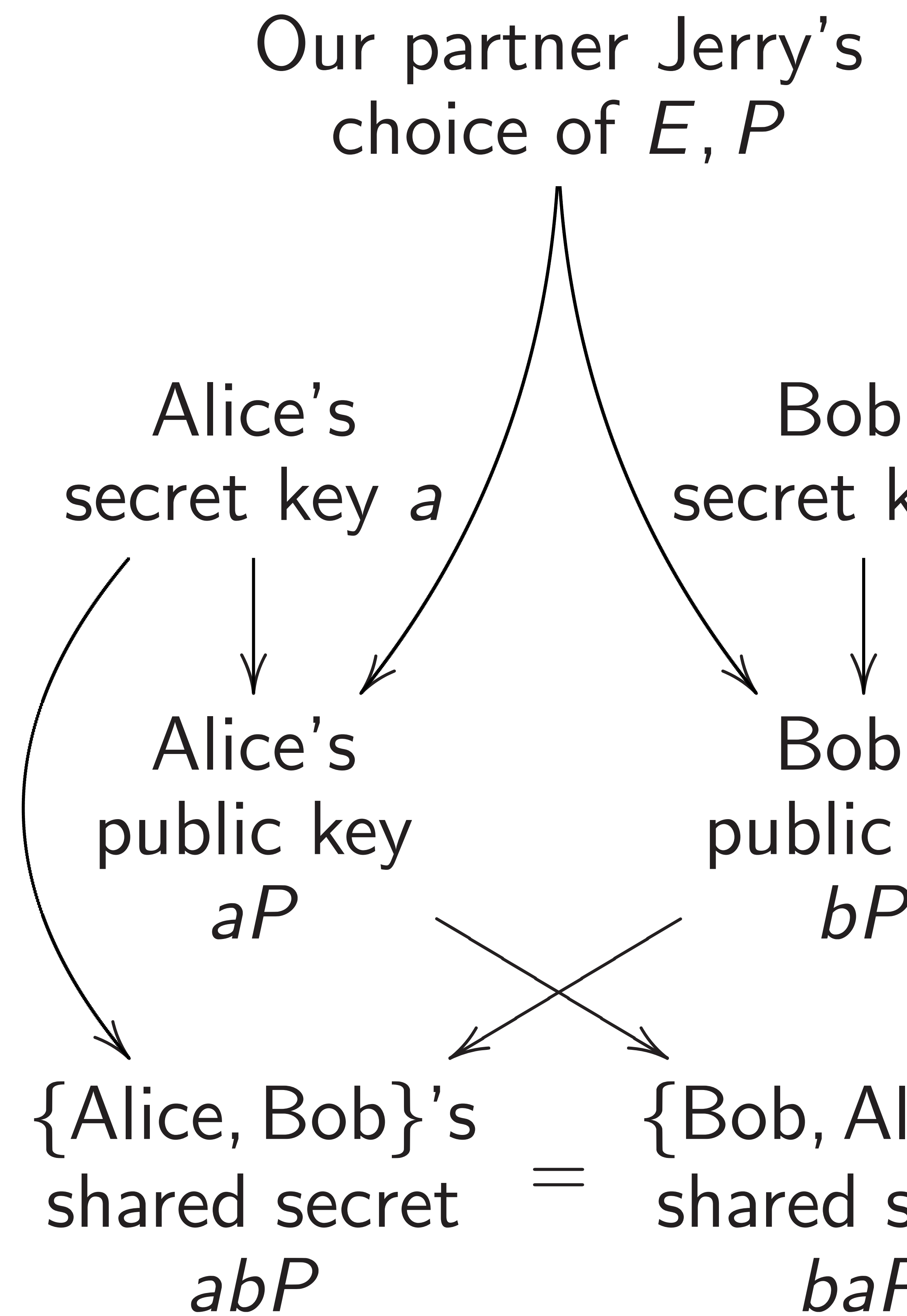


Textbook key exchange
using standard point $P$
on a standard elliptic curve $E$:



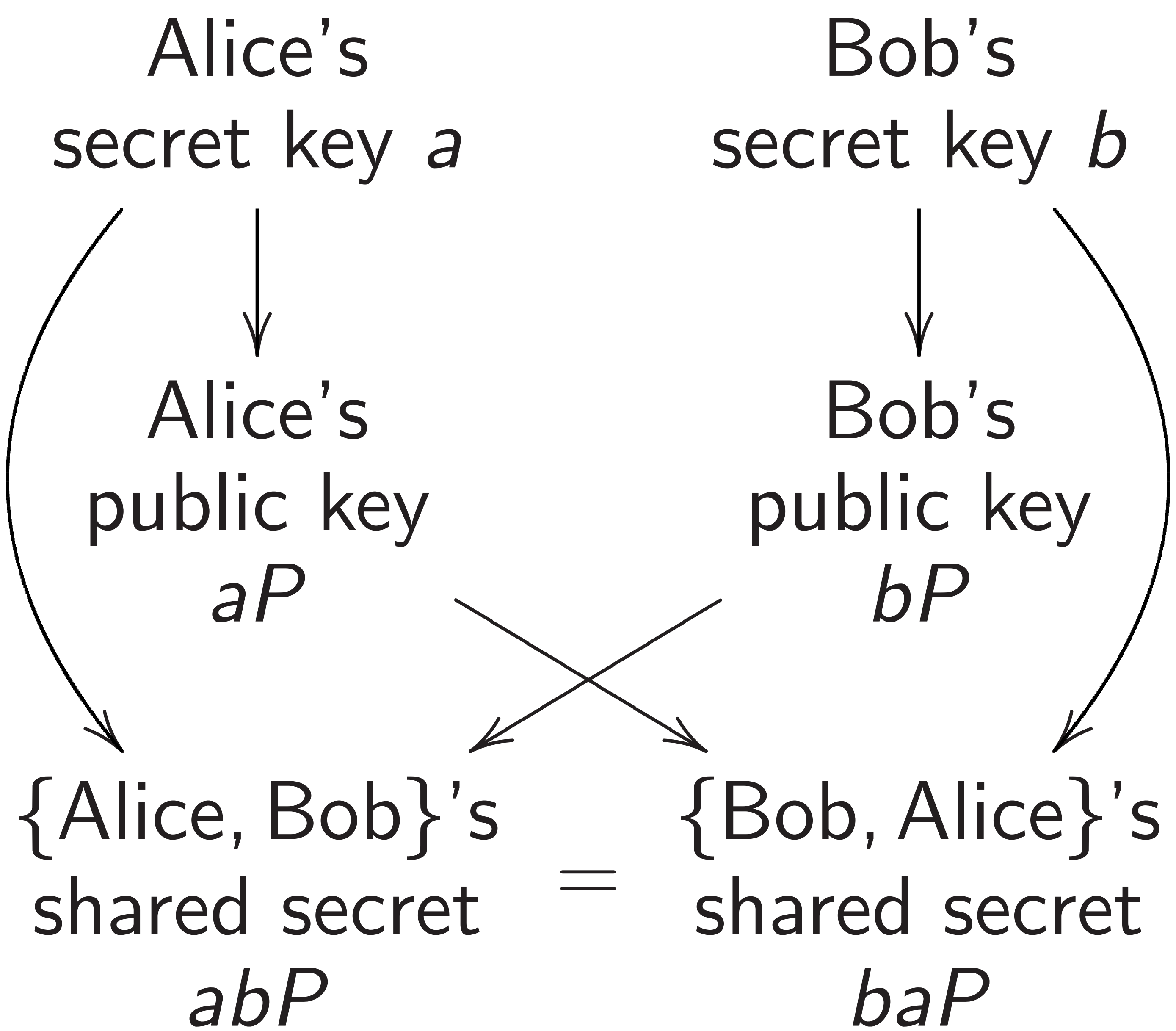

Security depends on choice of $E$.





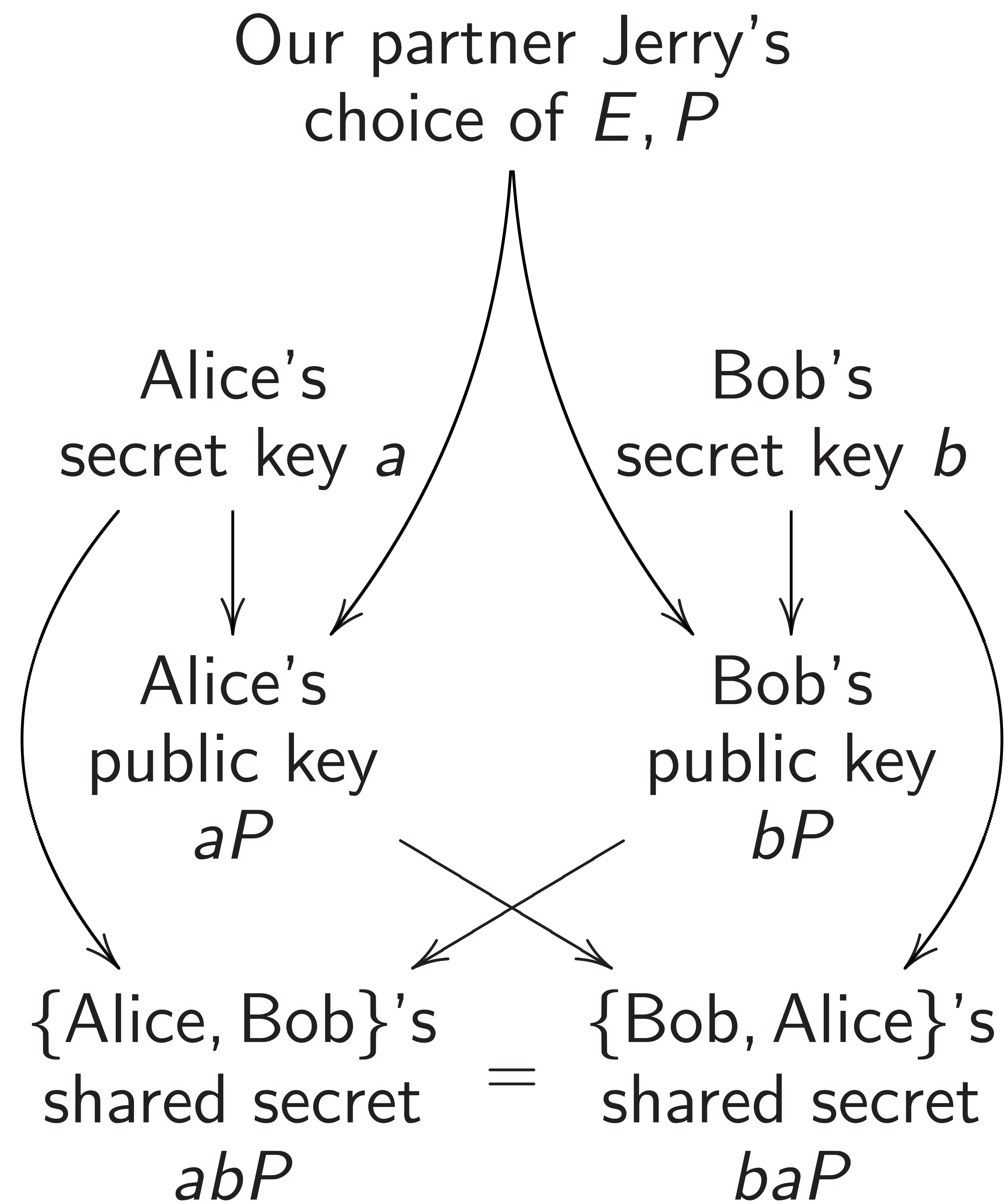

**This is not the same pictu**

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




Security depends on choice of $E$.



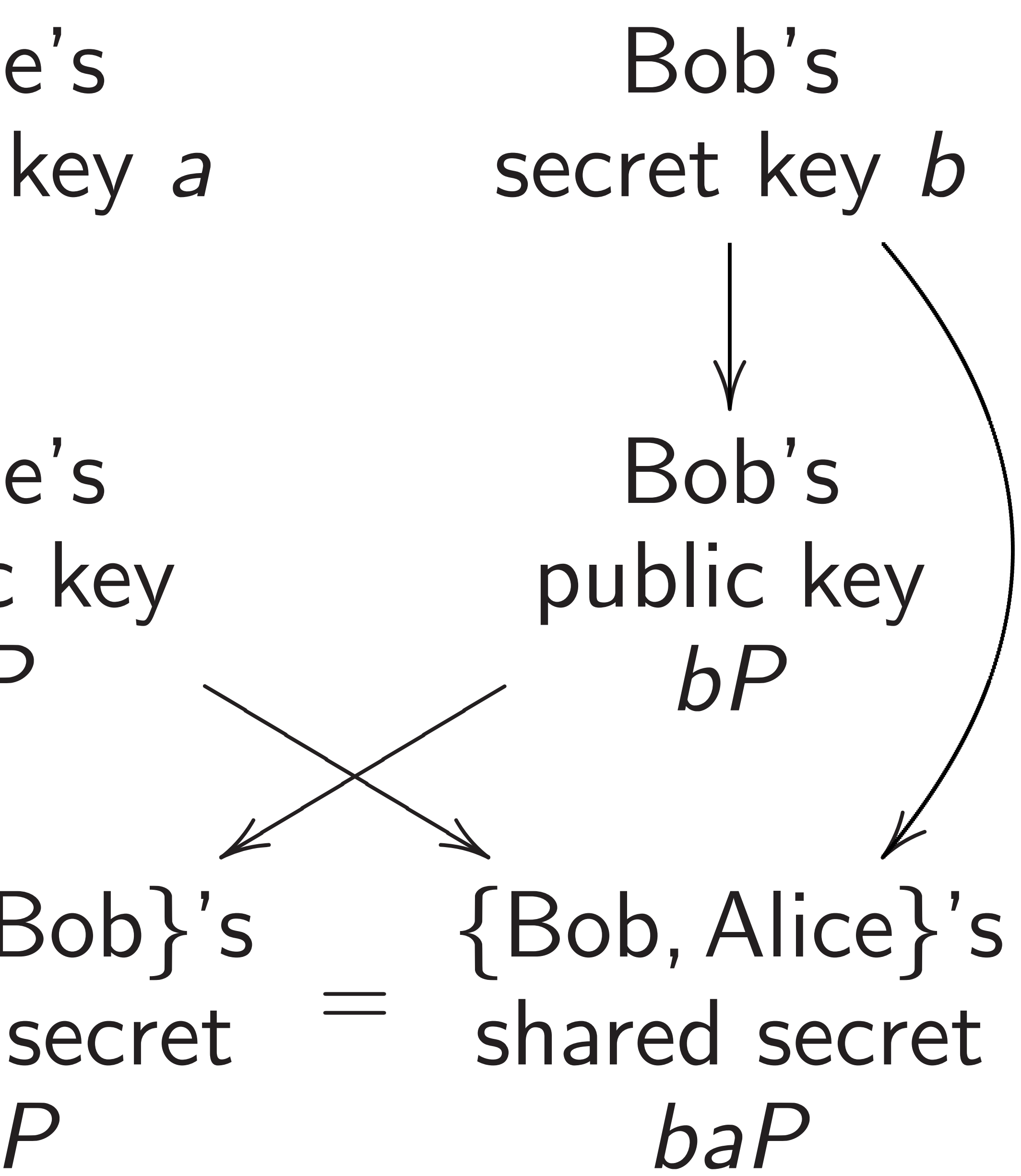



**This is not the same picture!**

k key exchange

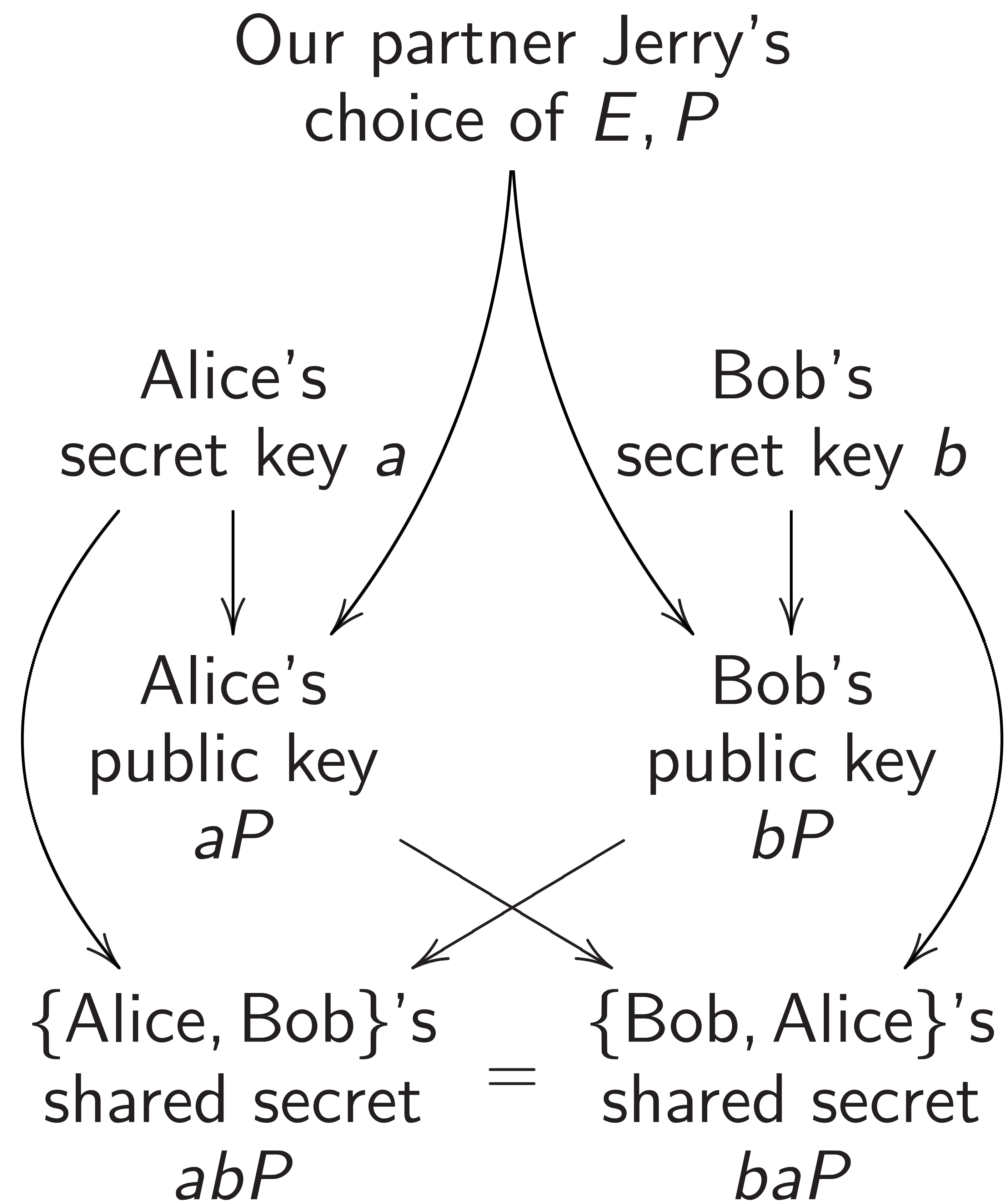
andard point $P$

ndard elliptic curve $E$:




depends on choice of $E$.






**This is not the same picture!**

One fina

2005 Br

"The ch

from wh

paramet

not mot

part of t

. . . **Veri**

The [Bra

generate

manner

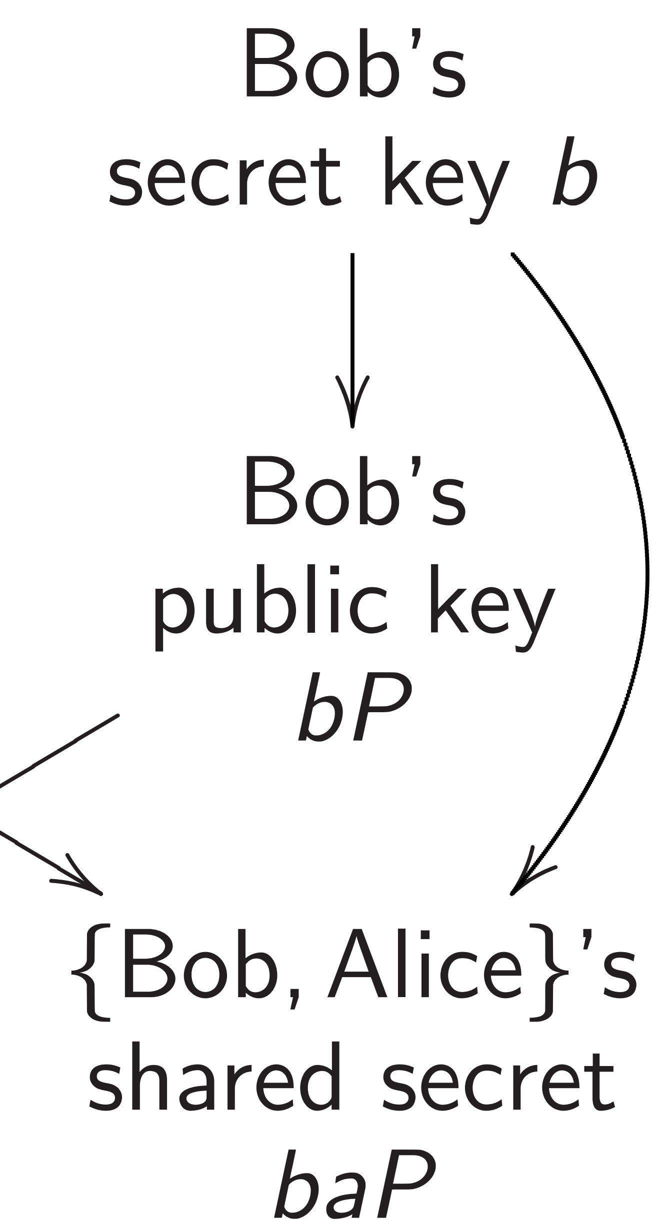
generate

compreh

hange

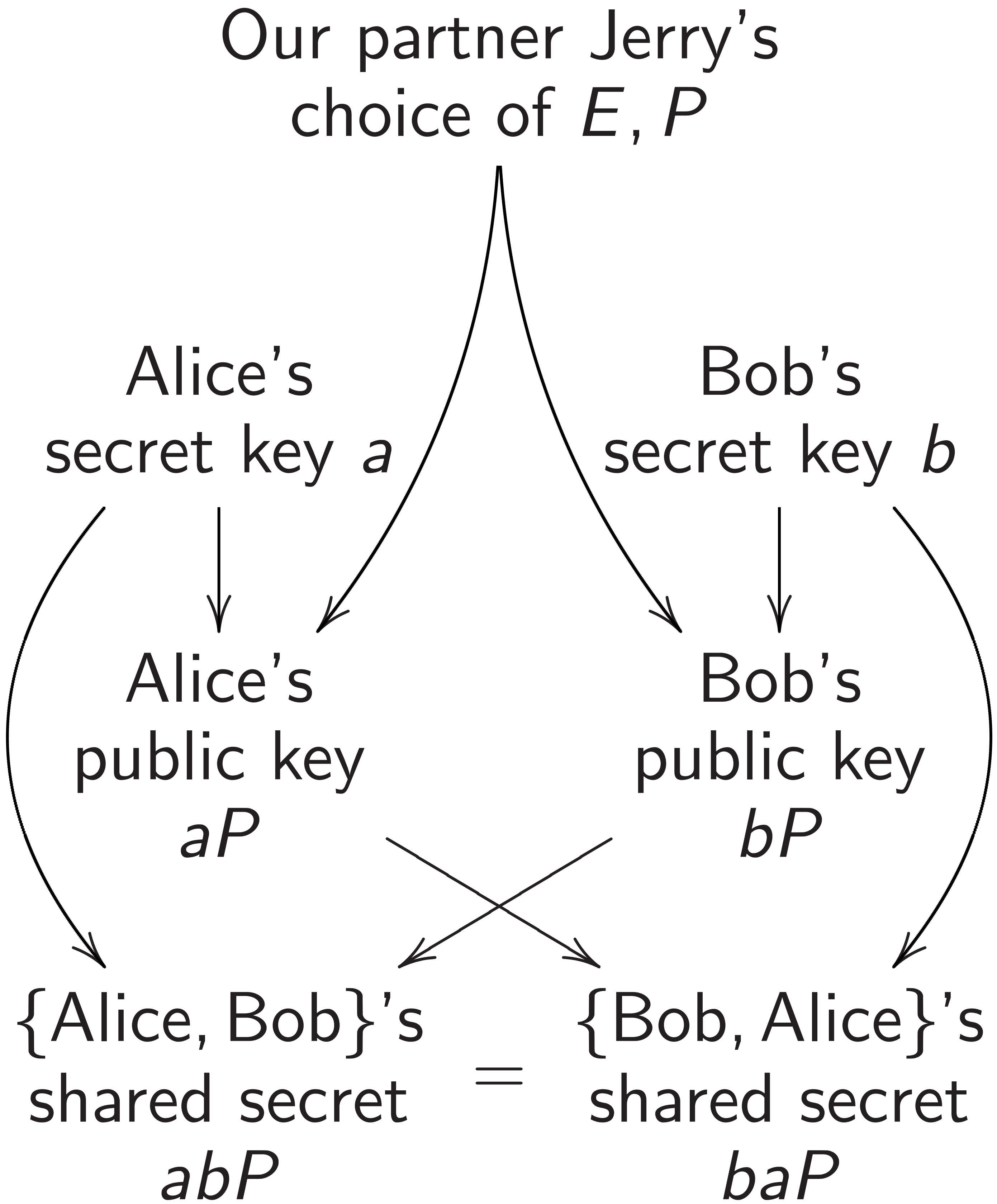
int $P$

tic curve $E$:




on choice of $E$.






**This is not the same picture!**

One final example

2005 Brainpool sta

"The choice of the

from which the [N

parameters have b

not motivated leav

part of the security

. . . **Verifiably pse**

The [Brainpool] cu

generated in a pse

manner using seed

generated in a syst

comprehensive way

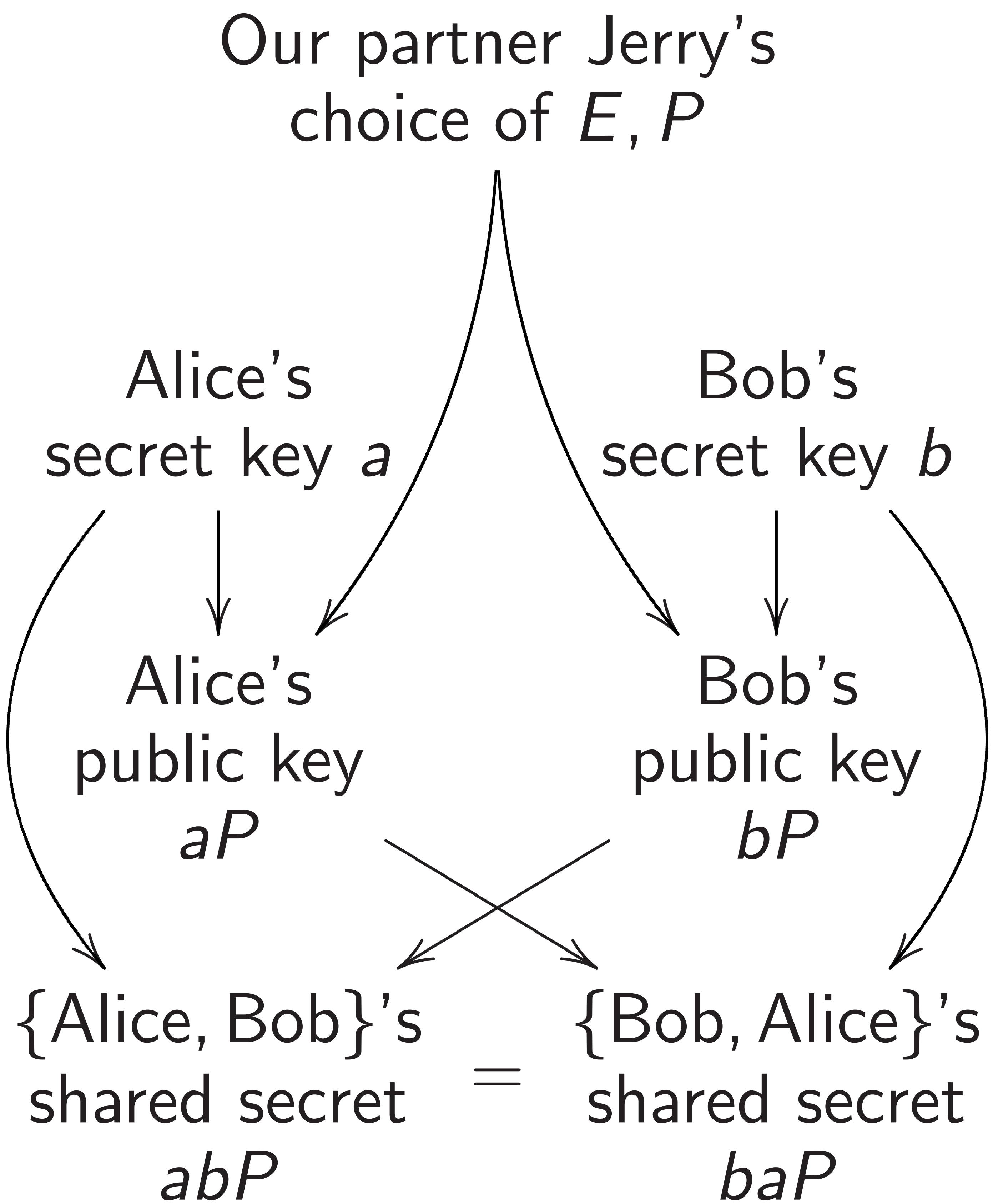
$E$:

's
key $b$

's
key

ice}'s
secret

of $E$.

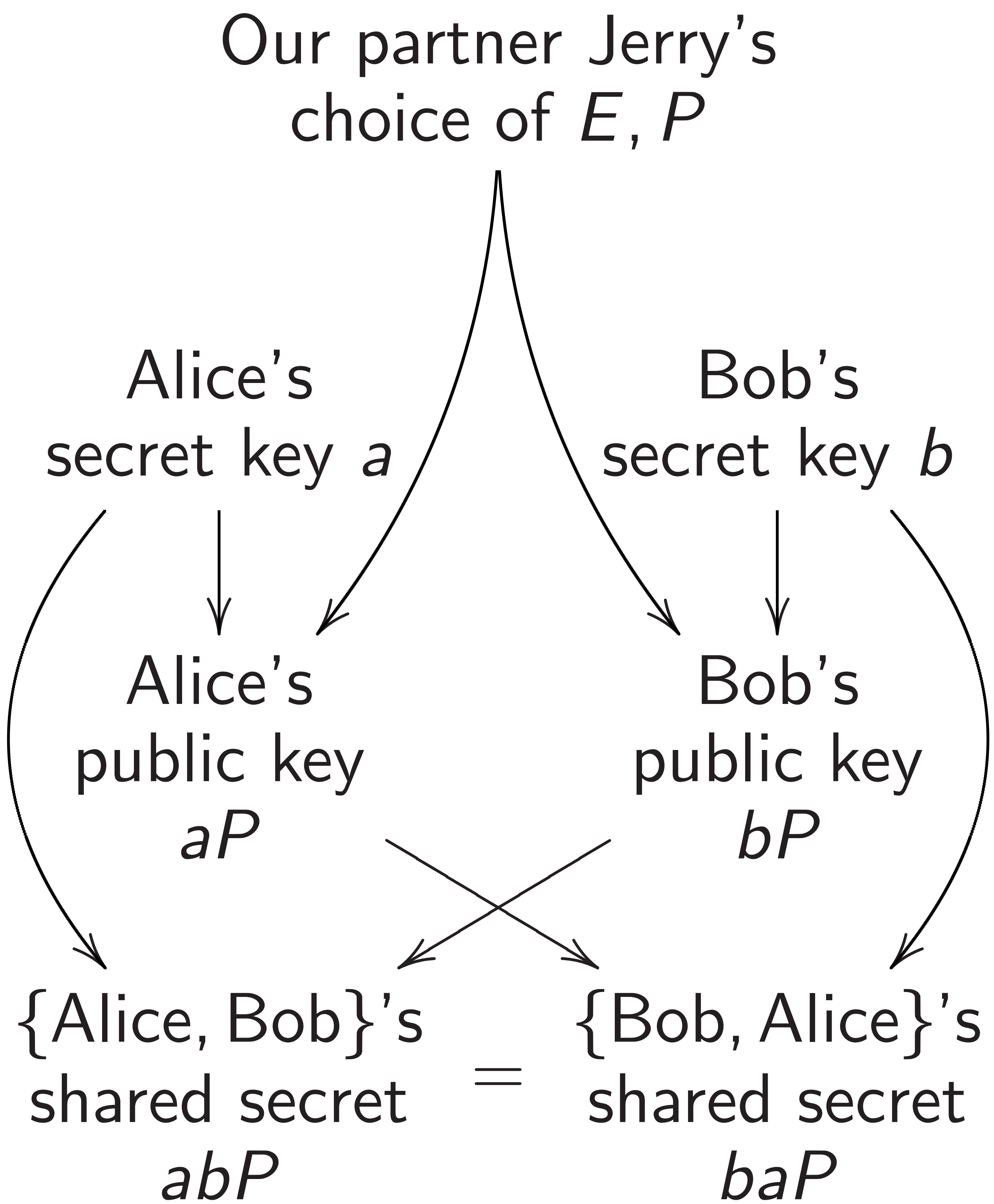



**This is not the same picture!**



One final example

2005 Brainpool standard:
"The choice of the seeds
from which the [NIST] curve
parameters have been derive
not motivated leaving an ess
part of the security analysis
. . . **Verifiably pseudo-rand**
The [Brainpool] curves shall
generated in a pseudo-rando
manner using seeds that are
generated in a systematic ar
comprehensive way."

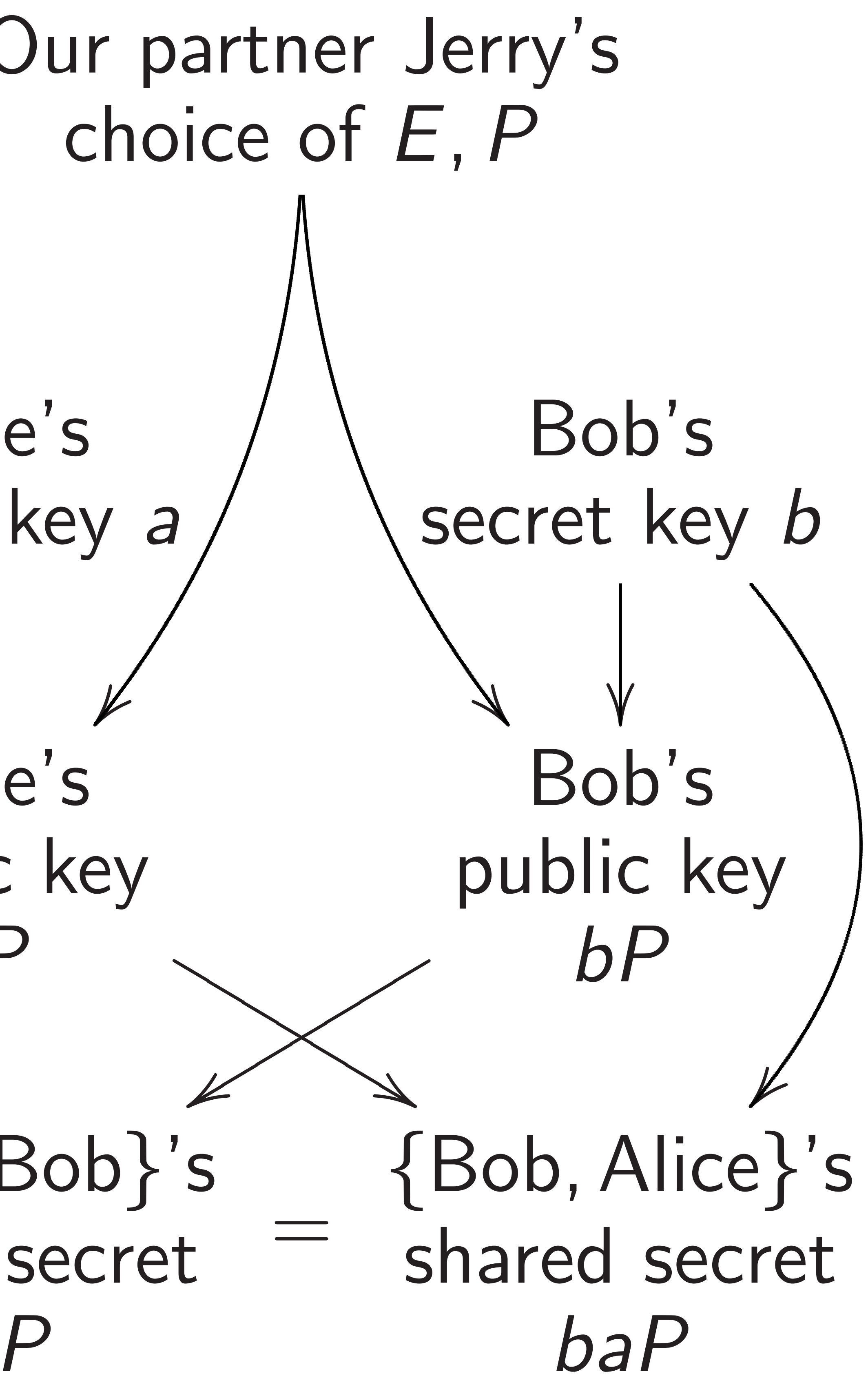



**This is not the same picture!**



## One final example

2005 Brainpool standard:
"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."




**not the same picture!**

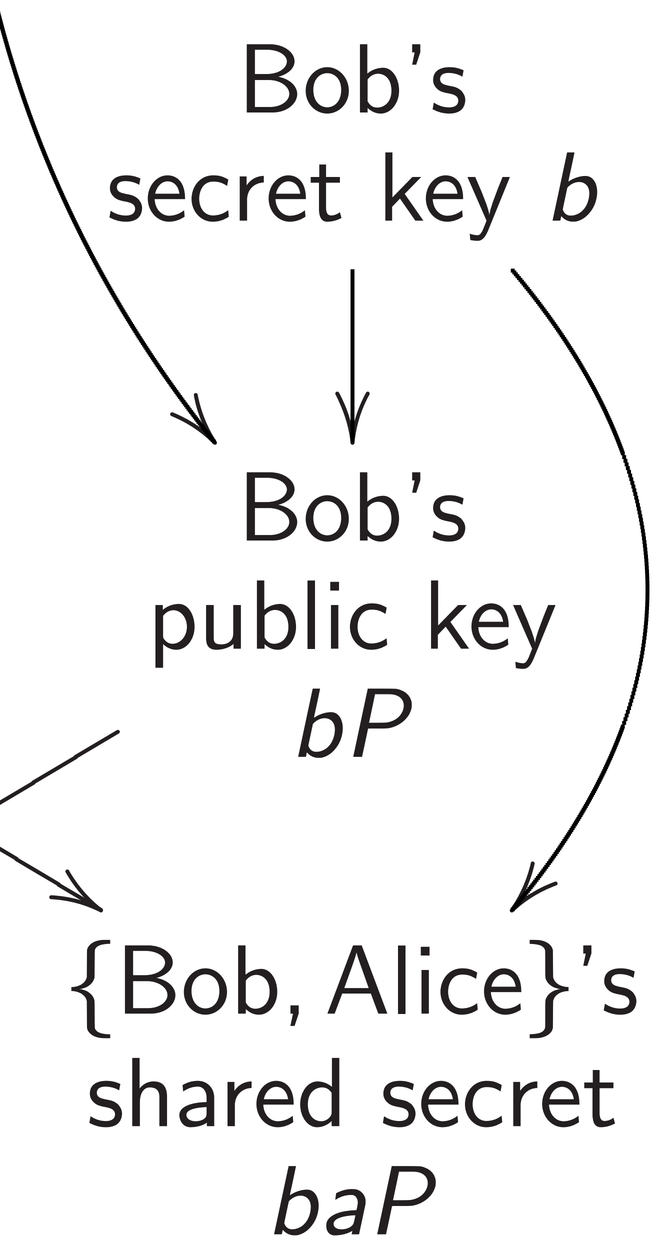


## One final example

2005 Brainpool standard: "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
```

er Jerry's
f $E, P$




**ame picture!**



## One final example

2005 Brainpool standard:
"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
```

’s
key $b$

’s
key

ice}’s
secret

**re!**



# One final example

2005 Brainpool standard:
“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
```

# One final example

2005 Brainpool standard:
“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
```

## l example

ainpool standard:
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
```

2015: W
the curv
from the
Previous

Output

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

andard:
e seeds
IST] curve
een derived is
ing an essential
y analysis open.
**eudo-random.**
urves shall be
udo-random
s 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
```

2015: We carefully
the curve-generati
from the Brainpoo
Previous slide: 22

Output of this pro

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

e

ed is

sential

open.

**lom.**

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
```

2015: We carefully impleme

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
```



2015: 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
```



2015: 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
```



2015: 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()
```



2015: 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
```



2015: 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))])

56^bytes)),bytes)
```



2015: 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
```

2015: 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
```

/e carefully 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

Brainpoo
advertise
“compre
transpar
say the s

y implemented
on procedure
l standard.
4-bit procedure.

cedure:

787B09F075797DA89F57EC8C0FF
9AE565C3253E8AEC4BFE84C659E
A2EBFA3870D98976FA2F17D2D8D

-bit Brainpool
**ame 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*

nted

ure

l.

edure.

57EC8C0FF

FE84C659E

2F17D2D8D

ool

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

```
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.

```
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 avoic
complica
hash out
from SH
maximur
Also upg
maximur

Brainpoo
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 NIS

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) = π
uses sin(1), so we
We also used muc
pattern of searchin

h.digest()

0FF

range(bytes))])

56^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 N 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?
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
r publication?
fore 2015?

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

ulate choice
r 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
tional

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.

l Brainpool's
ations of concatenating
tputs: We upgraded
IA-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.

o(1) = e
π/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).

ls.



```
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.