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.

The DES key size

IBM: 128! NSA: 32!

IBM: 64! NSA: 48!

Final compromise: 56.

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.

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!

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.

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.

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?

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.

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.

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.

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.

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!

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.

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!

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.

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.

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.

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.

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:

$$\begin{tikzcd}
\text{Alice's secret key } a \arrow[d] \arrow[dd, bend right] & & \text{Bob's secret key } b \arrow[d] \arrow[dd, bend left] \\
\text{Alice's public key } aP \arrow[drr] & & \text{Bob's public key } bP \arrow[dll] \\
\{\text{Alice, Bob}\}\text{'s shared secret } abP & = & \{\text{Bob, Alice}\}\text{'s shared secret } baP
\end{tikzcd}$$

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

Alice's secret key a

Bob's secret key b

Alice's public key aP

Bob's public key bP

$\{$Alice, Bob$\}$'s shared secret abP $=$ $\{$Bob, Alice$\}$'s shared secret baP

Security depends on choice of E.

Our partner Jerry's
choice of E, P

Alice's
secret key a

Bob's
secret key b

Alice's
public key
aP

Bob's
public key
bP

$\{$Alice, Bob$\}$'s
shared secret
abP

$=$

$\{$Bob, Alice$\}$'s
shared secret
baP

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."

```
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
```

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
```

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
```

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.

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

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

Can quietly manipulate choice to take the weaker procedure.

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

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

Can quietly manipulate choice to take the weaker procedure.

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

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

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

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

```
import simplesha3
hash = simplesha3.sha3512

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

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

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

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

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

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

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

Output: `7144BA12CE8A0C3BEFA053EDBADA55...`

Output: `7144BA12CE8A0C3BEFA053EDBADA55...`

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

Output: `7144BA12CE8A0C3BEFA053EDBADA55...`

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

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