NIST P-256 has a cube-root ECDL algorithm

D. J. Bernstein
University of Illinois at Chicago,
Technische Universiteit Eindhoven

Joint work with:
Tanja Lange
Technische Universiteit Eindhoven

---

eprint.iacr.org/2012/318,
eprint.iacr.org/2012/458:
“Non-uniform cracks in the concrete”, “Computing small discrete logarithms faster”

Central question:
What is the best ECDL algorithm for the NIST P-256 elliptic curve?

ECDL algorithm input:
curve point Q.

ECDL algorithm output: $\log_P Q$, where P is standard generator.

Standard definition of "best":
minimize "time".

Central question:
What is the best ECDL algorithm for the NIST P-256 elliptic curve?

ECDL algorithm input: curve point Q.

ECDL algorithm output: $\log_P Q$, where P is standard generator.

Standard definition of "best": minimize "time".

More generally, allow algorithms with $<$100% success probability; analyze tradeoffs between "time" and success probability.

Trivial standard conversion from any P-256 ECDL algorithm into (e.g.) signature-forgery attack against P-256 ECDSA:

- Use the ECDL algorithm to find the secret key.
- Run the signing algorithm on attacker's forged message.

Compared to ECDL algorithm, attack has practically identical speed and success probability.

Should P-256 ECDSA users
be worried about this?

Should P-256 ECDSA users
be worried about this?

No. Many ECC researchers
have tried and failed
to find good ECDL algorithms.

Should P-256 ECDSA users
be worried about this?

No. Many ECC researchers
have tried and failed
to find good ECDL algorithms.

Standard conjecture:
For each $p \in [0, 1]$,
each P-256 ECDL algorithm
with success probability $\geq p$
takes "time" $\geq 2^{128} p^{1/2}$.

Interlude regarding "time"

How much "time" does the following algorithm take?

```
def pidigit(n0,n1,n2):
  if n0 == 0:
    if n1 == 0:
      if n2 == 0: return 3
      return             1
    if n2 == 0: return   4
    return               1
  if n1 == 0:
    if n2 == 0: return   5
    return               9
  if n2 == 0: return     2
  return                 6
```

Students in algorithm courses
learn to count executed “steps”.
Skipped branches take 0 “steps”.

This algorithm uses 4 “steps”.

Students in algorithm courses
learn to count executed "steps".
Skipped branches take 0 "steps".

This algorithm uses 4 "steps".

Generalization: There exists an algorithm that, given $n < 2^k$, prints the nth digit of π using $k + 1$ "steps".

Students in algorithm courses learn to count executed "steps". Skipped branches take 0 "steps".

This algorithm uses 4 "steps".

Generalization: There exists an algorithm that, given $n < 2^k$, prints the nth digit of π using $k + 1$ "steps".

Variant: There exists a 259-"step" P-256 ECDL algorithm (with 100% success probability).

Students in algorithm courses learn to count executed “steps”. Skipped branches take 0 “steps”.

This algorithm uses 4 “steps”.

Generalization: There exists an algorithm that, given $n < 2^k$, prints the nth digit of π using $k + 1$ “steps”.

Variant: There exists a 259-“step” P-256 ECDL algorithm (with 100% success probability). If “time” means “steps” then the standard conjecture is wrong.

2000 Bellare–Kilian–Rogaway: "We fix some particular Random Access Machine (RAM) as a model of computation. . . . A's running time [means] A's actual execution time plus the length of A's description . . . This convention eliminates pathologies caused [by] arbitrarily large lookup tables . . . Alternatively, the reader can think of circuits over some fixed basis of gates, like 2-input NAND gates . . . now time simply means the circuit size."

Side comments:

1. Definition from Crypto 1994 Bellare–Kilian–Rogaway was flawed: failed to add length.

Paper conjectured "useful" DES security bounds; any reasonable interpretation of conjecture was false, given paper's definition.

Side comments:

1. Definition from Crypto 1994 Bellare–Kilian–Rogaway was flawed: failed to add length.

Paper conjectured “useful” DES security bounds; any reasonable interpretation of conjecture was false, given paper’s definition.

2. Many more subtle issues defining RAM “time”: see 1990 van Emde Boas survey.

Side comments:

1. Definition from Crypto 1994 Bellare–Kilian–Rogaway was flawed: failed to add length.

Paper conjectured “useful” DES security bounds; any reasonable interpretation of conjecture was false, given paper’s definition.

2. Many more subtle issues defining RAM “time”: see 1990 van Emde Boas survey.

3. NAND definition is easier but breaks many theorems.

Two-way reductions

Another standard conjecture:
For each $p \in [2^{-40}, 1]$,
each P-256 ECDSA attack
with success probability $\geq p$
takes "time" $> 2^{128} p^{1/2}$.

Two-way reductions

Another standard conjecture:
For each $p \in [2^{-40}, 1]$,
each P-256 ECDSA attack
with success probability $\geq p$
takes "time" $> 2^{128} p^{1/2}$.

Why should users have any
confidence in this conjecture?

How many ECC researchers have
really tried to break ECDSA?
ECDH? Other ECC protocols?
Far less attention than for ECDL.

Provable security to the rescue!

Prove: if there is an ECDSA attack then there is an ECDL attack with similar “time” and success probability.

Provable security to the rescue!

Prove: if there is an ECDSA attack then there is an ECDL attack with similar "time" and success probability.

Oops: This turns out to be hard. But changing from ECDSA to Schnorr allows a proof: Eurocrypt 1996 Pointcheval–Stern.

Provable security to the rescue!

Prove: if there is an ECDSA attack then there is an ECDL attack with similar “time” and success probability.

Oops: This turns out to be hard. But changing from ECDSA to Schnorr allows a proof: Eurocrypt 1996 Pointcheval–Stern.

Oops: This proof has very bad “tightness” and is only for limited classes of attacks. Continuing efforts to fix these limitations.

Similar pattern throughout the "provable security" literature.

Protocol designers (try to) prove that hardness of a problem P (e.g., the ECDL problem) implies security of various protocols Q.

After extensive cryptanalysis of P, maybe gain confidence in hardness of P, and hence in security of Q.

Similar pattern throughout the "provable security" literature.

Protocol designers (try to) prove that hardness of a problem P (e.g., the ECDL problem) implies security of various protocols Q.

After extensive cryptanalysis of P, maybe gain confidence in hardness of P, and hence in security of Q.

Why not directly cryptanalyze Q? Cryptanalysis is hard work: have to focus on *a few* problems P. Proofs scale to *many* protocols Q.


CLINT EASTWOOD
THE GOOD
THE BAD
AND THE UGLY
2-DISC DVD COLLECTOR'S SET

Have cryptanalysts actually
studied the problem P
that the protocol designer
hypothesizes to be hard?

Have cryptanalysts actually studied the problem P that the protocol designer hypothesizes to be hard?

Three different situations:

"The good": Cryptanalysts have studied P.

Have cryptanalysts actually studied the problem P that the protocol designer hypothesizes to be hard?

Three different situations:

"The good": Cryptanalysts have studied P.

"The bad": Cryptanalysts have not studied P.

Have cryptanalysts actually studied the problem P that the protocol designer hypothesizes to be hard?

Three different situations:

"The good": Cryptanalysts have studied P.

"The bad": Cryptanalysts have not studied P.

"The ugly": People *think* that cryptanalysts have studied P, but actually they've studied $P' \neq P$.

Cube-root ECDL algorithms

Assuming plausible heuristics, overwhelmingly verified by computer experiment:

There exists a P-256 ECDL algorithm that takes "time" $\approx 2^{85}$ and has success probability ≈ 1.

"Time" includes algorithm length.
"$\approx$": details later in the talk.

Inescapable conclusion: **The standard conjectures** (regarding P-256 ECDL hardness, P-256 ECDSA security, etc.) **are false.**

Switch to P-384 but
continue using 256-bit scalars?

Switch to P-384 but continue using 256-bit scalars?

Doesn't fix the problem.
There exists a P-384 ECDL algorithm that takes "time" $\approx 2^{85}$ and has success probability ≈ 1 for P, Q with 256-bit $\log_P Q$.

Switch to P-384 but
continue using 256-bit scalars?

Doesn't fix the problem.
There exists a P-384 ECDL
algorithm that takes "time" $\approx 2^{85}$
and has success probability ≈ 1
for P, Q with 256-bit $\log_P Q$.

To push the cost of these attacks
up to 2^{128}, switch to P-384
and switch to 384-bit scalars.
This is not common practice:
users don't like $\approx 3\times$ slowdown.

DON'T PANIC

Should P-256 ECDSA users be worried about this P-256 ECDL algorithm A?

No!

We have a program B that prints out A, but B takes "time" $\approx 2^{170}$.

We conjecture that nobody will ever print out A.

Should P-256 ECDSA users
be worried about this
P-256 ECDL algorithm A?

No!

We have a program B
that prints out A,
but B takes "time" $\approx 2^{170}$.

We conjecture that
nobody will ever print out A.

But A *exists*, and the standard
conjecture doesn't see the 2^{170}.

Cryptanalysts *do* see the 2^{170}.

Common parlance: We have a 2^{170} "precomputation" (independent of Q) followed by a 2^{85} "main computation".

For cryptanalysts: This costs 2^{170}, much worse than 2^{128}.

For the standard security definitions and conjectures: The main computation costs 2^{85}, much better than 2^{128}.

What the algorithm does

What the algorithm does

1999 Escott–Sager–Selkirk–Tsapakidis, also crediting Silverman–Stapleton:

Computing (e.g.) $\log_P Q_1$, $\log_P Q_2$, $\log_P Q_3$, $\log_P Q_4$, and $\log_P Q_5$ costs only $2.49\times$ more than computing $\log_P Q$.

The basic idea:
compute $\log_P Q_1$ with rho;
compute $\log_P Q_2$ with rho, *reusing* distinguished points produced by Q_1; etc.

2001 Kuhn–Struik analysis:
cost $\Theta(n^{1/2}\ell^{1/2})$
for n discrete logarithms
in group of order ℓ
if $n \ll \ell^{1/4}$.

2001 Kuhn–Struik analysis:
cost $\Theta(n^{1/2}\ell^{1/2})$
for n discrete logarithms
in group of order ℓ
if $n \ll \ell^{1/4}$.

2004 Hitchcock–
Montague–Carter–Dawson:
View computations of
$\log_P Q_1, \ldots, \log_P Q_{n-1}$ as
precomputatation for main
computation of $\log_P Q_n$.
Analyze tradeoffs between
main-computation time and
precomputation time.

2012 Bernstein–Lange:

(1) Adapt to interval of length ℓ inside much larger group.

(2) Analyze tradeoffs between main-computation time and precomputed table size.

(3) Choose table entries more carefully to reduce main-computation time.

(4) Also choose iteration function more carefully.

(5) Reduce space required for each table entry.

(6) Break $\ell^{1/4}$ barrier.

Applications:

(7) Disprove the standard 2^{128}
P-256 security conjectures.

(8) Accelerate trapdoor DL etc.

(9) Accelerate BGN etc.;
this needs (1).

Bonus:

(10) Disprove the standard 2^{128}
AES, DSA-3072, RSA-3072
security conjectures.

Credit to earlier Lee–Cheon–Hong
paper for (2), (6), (8).

The basic algorithm:

Precomputation:
Start some walks at yP
for random choices of y.
Build table of distinct
distinguished points D
along with $\log_P D$.

Main computation:
Starting from Q, walk to
distinguished point $Q + yP$.
Check for $Q + yP$ in table.

(If this fails, rerandomize Q.)

Standard walk function:
choose uniform random
$c_1, \ldots, c_r \in \{1, 2, \ldots, \ell - 1\}$;
walk from R to $R + c_{H(R)}P$.

Nonstandard tweak:
reduce $\ell - 1$ to, e.g., $0.25\ell/W$,
where W is average walk length.

Intuition: This tweak
compromises performance by
only a small constant factor.

Standard walk function: choose uniform random $c_1, \ldots, c_r \in \{1, 2, \ldots, \ell - 1\}$; walk from R to $R + c_{H(R)}P$.

Nonstandard tweak: reduce $\ell - 1$ to, e.g., $0.25\ell/W$, where W is average walk length.

Intuition: This tweak compromises performance by only a small constant factor.

If tweaked algorithm works for a group of order ℓ, what will it do for an interval of order ℓ?

Are rho and kangaroo really so different? Seek unification: “kangarho”?

Are rho and kangaroo really so different? Seek unification: "kangarho"? Not approved by coauthor: "kangarhoach"?

Are rho and kangaroo really so different? Seek unification: "kangarho"? Not approved by coauthor: "kangarhoach"?

Some of our experiments for average ECDL computations using table of size $\approx\ell^{1/3}$ (selected from somewhat larger table):
for group of order ℓ,
precomputation $\approx 1.24\ell^{2/3}$,
main computation $\approx 1.77\ell^{1/3}$;
for interval of order ℓ,
precomputation $\approx 1.21\ell^{2/3}$,
main computation $\approx 1.93\ell^{1/3}$.

Interlude: constructivity

Bolzano–Weierstrass theorem:
every sequence $x_0, x_1, \ldots \in [0, 1]$
has a converging subsequence.

The standard proof:

Define $I_1 = [0, 0.5]$
if $[0, 0.5]$ has infinitely many x_i;
otherwise define $I_1 = [0.5, 1]$.
Define I_2 similarly
as left or right half of I_1; etc.

Take smallest i_1 with $x_{i_1} \in I_1$,
smallest $i_2 > i_1$ with $x_{i_2} \in I_2$,
etc.

Kronecker's reaction: WTF?

Kronecker's reaction: WTF?

This is not constructive.

This proof gives us no way to *find* I_1, even if each x_i is completely explicit.

Kronecker's reaction: WTF?
This is not constructive.
This proof gives us no way
to *find* I_1, even if each x_i
is completely explicit.

Early 20th-century formalists:
This objection is meaningless.
The only formalization of "one
can find x such that $p(x)$" is
"there exists x such that $p(x)$".

Kronecker's reaction: WTF?
This is not constructive.
This proof gives us no way
to *find* I_1, even if each x_i
is completely explicit.

Early 20th-century formalists:
This objection is meaningless.
The only formalization of "one
can find x such that $p(x)$" is
"there exists x such that $p(x)$".

Constructive mathematics later
introduced other possibilities,
giving a formal meaning
to Kronecker's objection.

Findable algorithms

"Time"-2^{170} algorithm B prints "time"-2^{85} ECDL algorithm A.

First attempt to formally quantify unfindability of A:
"What is the lowest cost for an algorithm that prints A?"

Findable algorithms

"Time"-2^{170} algorithm B prints "time"-2^{85} ECDL algorithm A.

First attempt to formally quantify unfindability of A:
"What is the lowest cost for an algorithm that prints A?"

Oops: This cost is 2^{85}, not 2^{170}.

Findable algorithms

"Time"-2^{170} algorithm B prints "time"-2^{85} ECDL algorithm A.

First attempt to formally quantify unfindability of A:
"What is the lowest cost for an algorithm that prints A?"

Oops: This cost is 2^{85}, not 2^{170}.

Our proposed quantification:
"What is the lowest cost for a *small* algorithm that prints A?"

Can consider longer chains:
A'' prints A' prints A.

The big picture

The literature on provable concrete security is full of security definitions that consider *all* "time $\leq T$" algorithms.

Cryptanalysts actually focus on a subset of these algorithms.

Widely understood for decades: this drastically changes cost of hash collisions.
Not widely understood: this drastically changes cost of breaking P-256, cost of breaking RSA-3072, etc.

What to do about this gap?

What to do about this gap?

Nitwit formalists: "Oops,
P-256 doesn't have 2^{128} security?
Thanks. New conjecture:
P-256 has 2^{85} security."

What to do about this gap?

Nitwit formalists: “Oops,
P-256 doesn’t have 2^{128} security?
Thanks. New conjecture:
P-256 has 2^{85} security.”

Why should users have any
confidence in this conjecture?

How many ECC researchers
have really tried to break it?

What to do about this gap?

Nitwit formalists: “Oops,
P-256 doesn’t have 2^{128} security?
Thanks. New conjecture:
P-256 has 2^{85} security.”

Why should users have any
confidence in this conjecture?

How many ECC researchers
have really tried to break it?

Why should cryptanalysts
study algorithms that
attackers can’t possibly use?

Much better answer:

Aim for unification of
(1) set of algorithms
feasible for attackers,
(2) set of algorithms
considered by cryptanalysts,
(3) set of algorithms
considered in definitions,
conjectures, theorems, proofs.

A gap between (1) and (3)
is a flaw in the definitions,
undermining the credibility
of provable security.

Adding uniformity
(i.e., requiring attacks to
work against many systems)
would increase the gap,
so we recommend against it.

We recommend

- adding findability and
- switching from “time” to price-performance ratio for chips (see, e.g., 1981 Brent–Kung).

Each recommendation kills the 2^{85} ECDL algorithm.