Introduction to Cryptography
2WF80
Discrete Logarithms

Tanja Lange
Technische Universiteit Eindhoven

08 December 2016

Diffie–Hellman key exchange

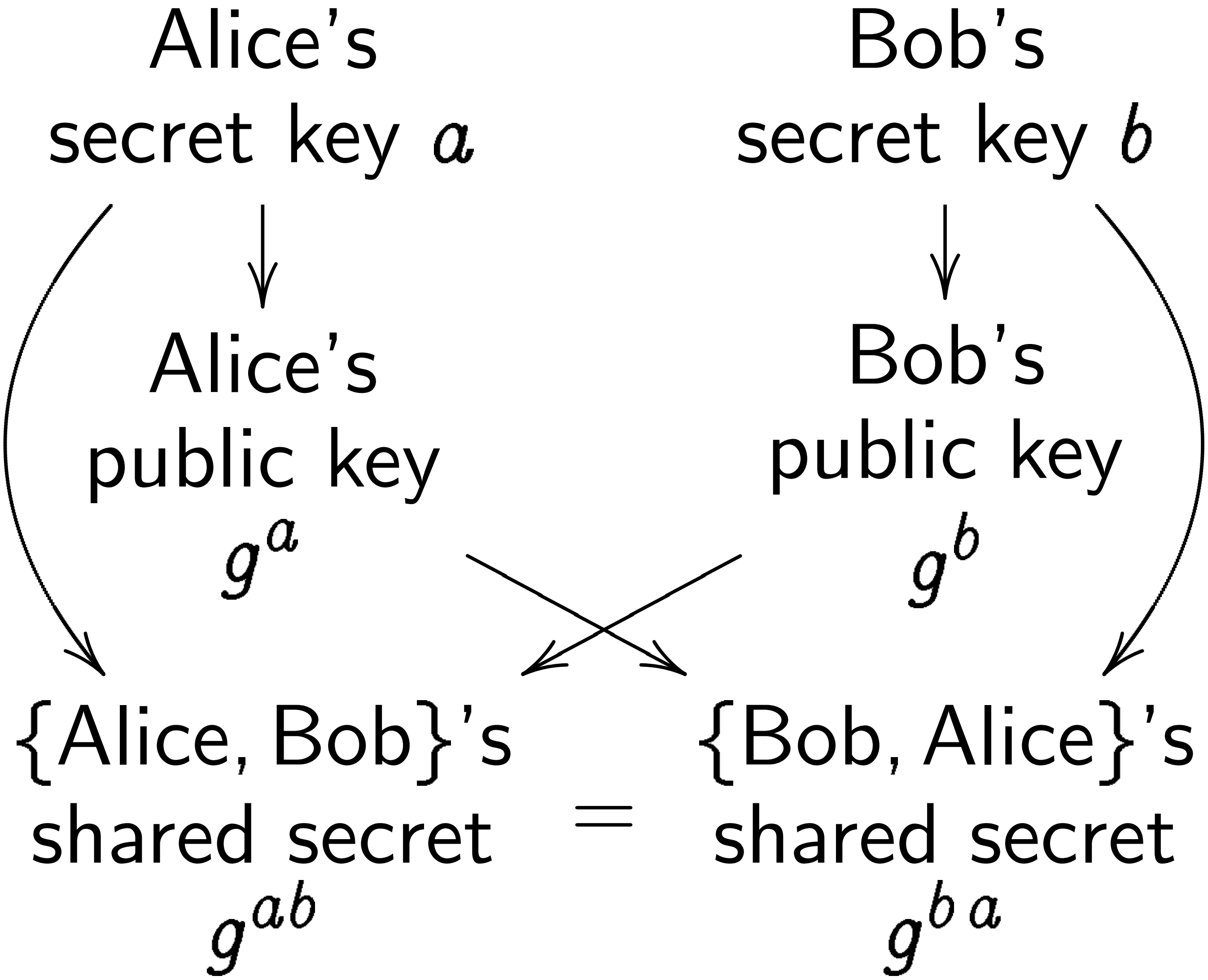
1976, first to introduce public-key cryptography.

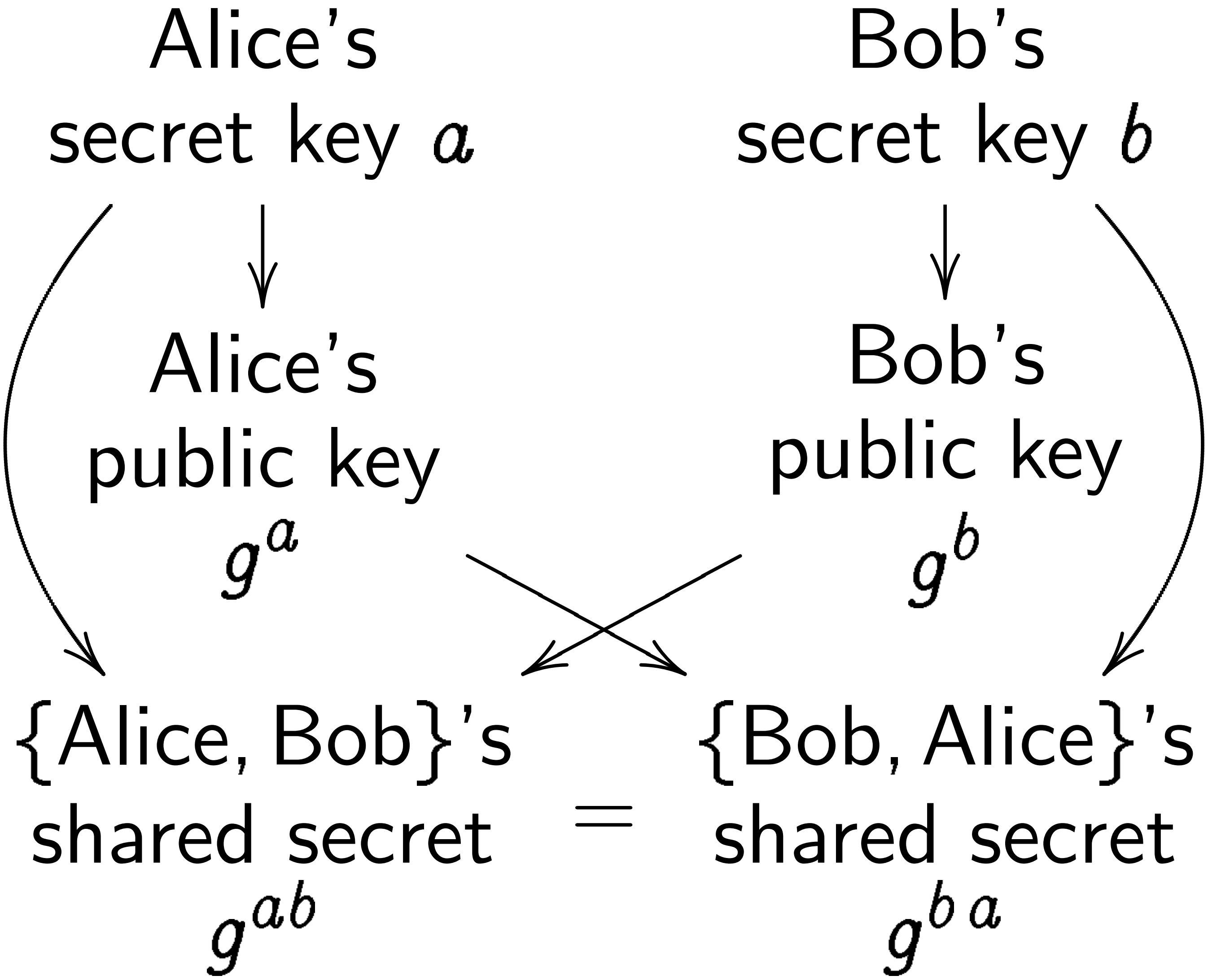
Standardize group G, & pick some $g \in G$.

Alice chooses secret a, computes her public key g^a.

Bob chooses secret b, computes his public key g^b.

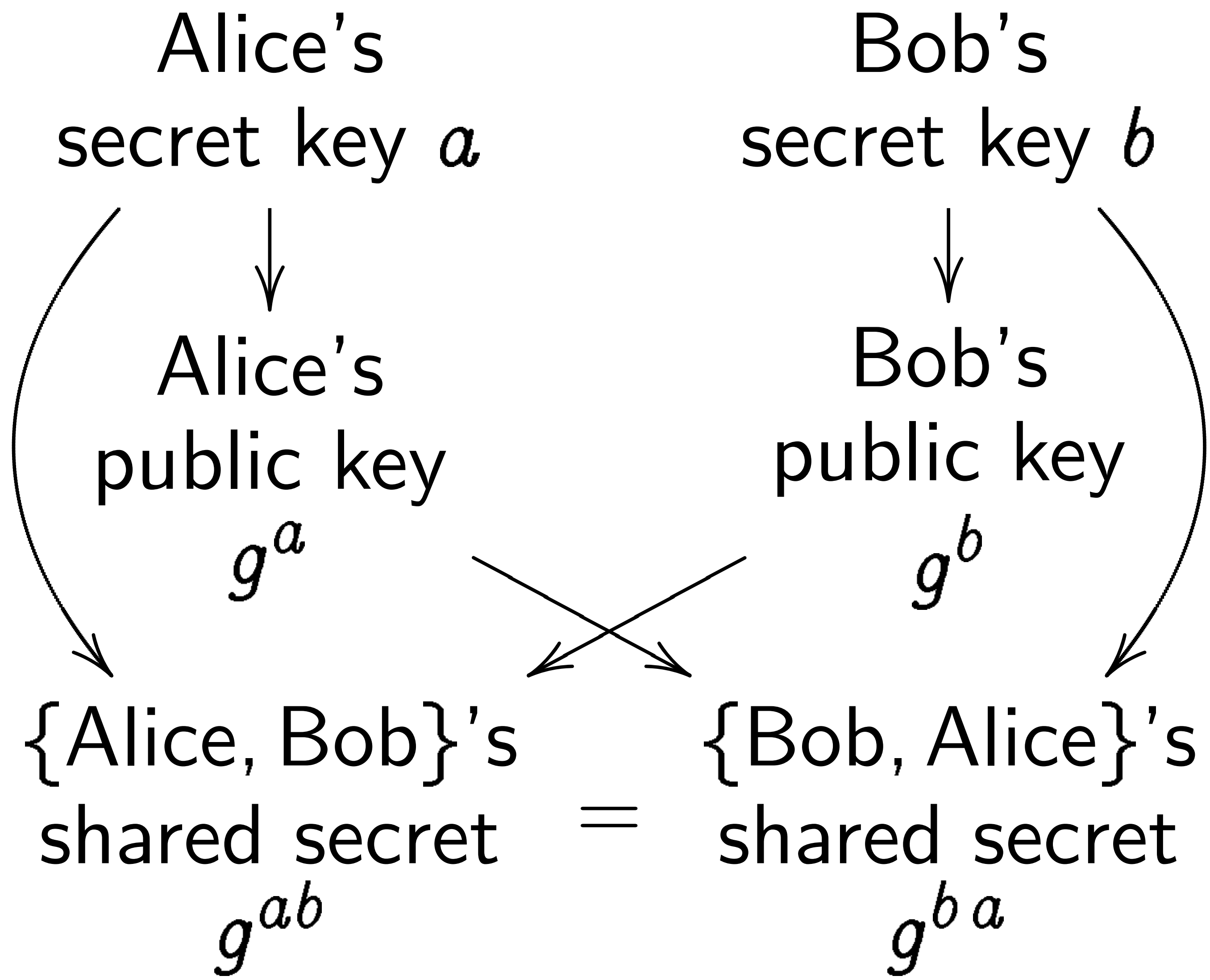
Alice computes $(g^b)^a$.
Bob computes $(g^a)^b$.
They use this shared secret to encrypt with symmetric crypto.


Alice's secret key a
Bob's secret key b
Alice's public key g^a
Bob's public key g^b
{Alice, Bob}'s shared secret g^{ab}
=
{Bob, Alice}'s shared secret g^{ba}



Warning #1: Many G are unsafe!

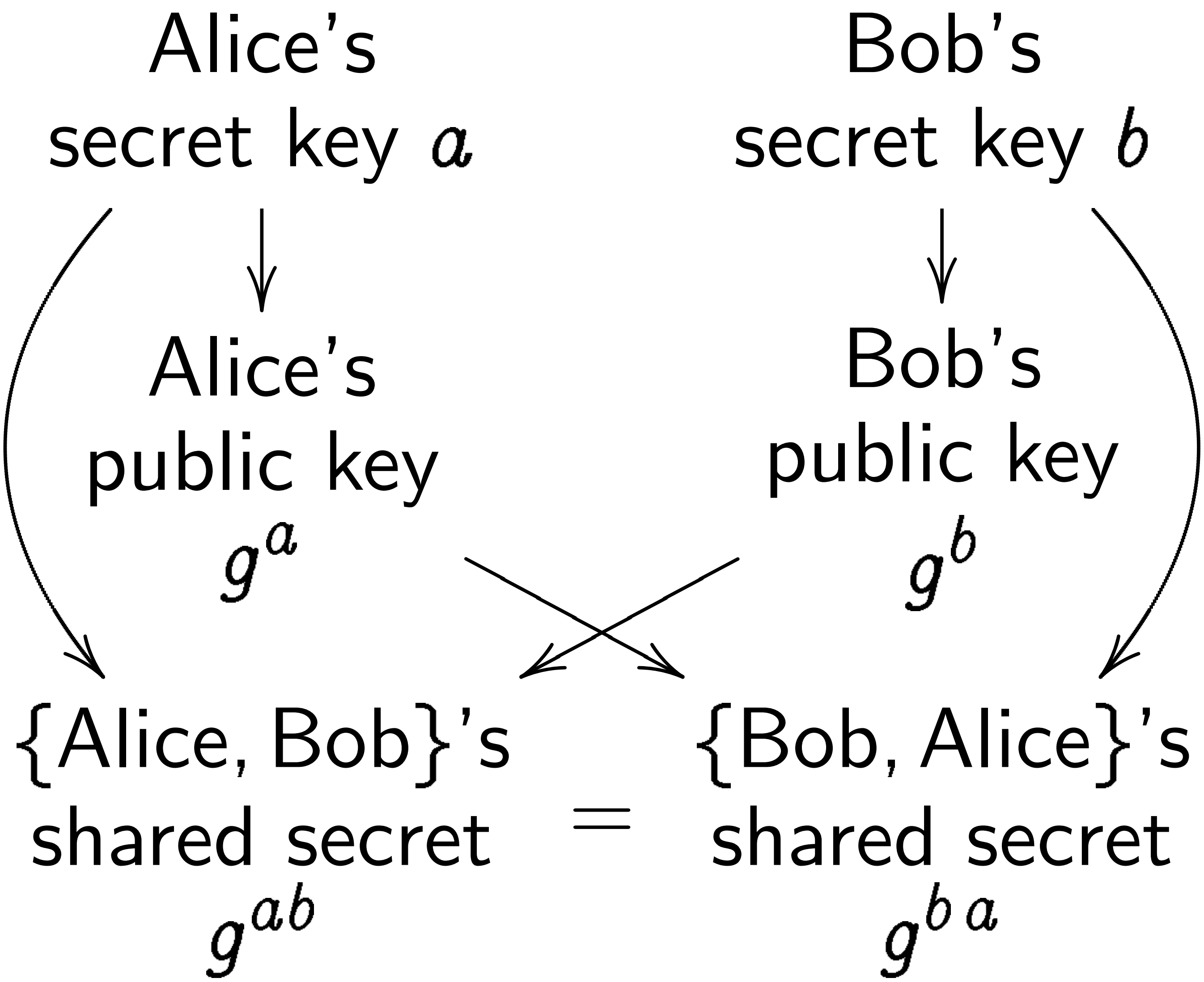
e.g. $G = \mathbf{Q}$, $g = 2$



Warning #1: Many G are unsafe!

e.g. $G = \mathbf{Q}$, $g = 2$ size gives away exponent.

$G = (\mathbf{F}_p, +)$, i.e., A sends ag.



Warning #1: Many G are unsafe!

e.g. $G = \mathbf{Q}$, $g = 2$ size gives away exponent.

$G = (\mathbf{F}_p, +)$, i.e., A sends ag.
E computes $a \equiv ag \cdot g^{-1} \bmod p$ using XGCD.

Diffie–Hellman key exchange

The proper DH proposal:

Standardize large prime p &
generator g of $\mathbf{F}_p^*$.

Alice chooses big secret $a < p-1$,
computes her public key g^a.

Bob chooses big secret b,
computes his public key g^b.

Alice computes $(g^b)^a$.
Bob computes $(g^a)^b$.
They use this shared secret
to encrypt with symmetric crypto.

Is this secure?

Computational Diffie-Hellman Problem (CDHP):
Given g, g^a, g^b
compute g^{ab}.

Decisional Diffie-Hellman Problem (DDHP):
Given g, g^a, g^b, and g^c
decide whether $g^c = g^{ab}$.

Discrete Logarithm Problem (DLP):
Given g, g^a, compute a.

If one can solve DLP, then CDHP and DDHP are easy.

Practical problems

Eve can set up a *man-in-the-middle* attack:

$$A \overset{g^{ae}}{\longleftrightarrow} E \overset{g^{bf}}{\longleftrightarrow} B$$

E decrypts everything from A and reencrypts it to B and vice versa.

This attack cannot be detected unless A and B have some long-term secrets.

Semi-static DH

Alice publishes long-term
public key g^a,
keeps long-term secret key a.

Any user can encrypt to
Alice using this key:
Pick random k, compute $r = g^k$
and encrypt message using key
derived from $(g^a)^k$.
Send ciphertext c along with r.

Alice decrypts, by obtaining
same key from $r^a = g^{ak}$.

ElGamal encryption

(For historical purposes only)

Alice publishes long-term
public key g^a,
keeps long-term secret key a.

Any user can encrypt to
Alice using this key:
Pick random k, compute $r = g^k$.
Encrypt $m \in \mathbf{F}_p^*$ as $c = (g^a)^k \cdot m$.
Send (r, c).

Alice decrypts, by computing
$m = c/(r^a) = (g^a)^k \cdot m/g^{ak}$.

Downside: requires m in group;
has multiplicative structure.

ElGamal signatures

Requires a hash function.

Let $g \in \mathbf{F}_p^*$ have prime order ℓ.

Alice publishes long-term
public key g^a,
keeps long-term secret key a.

Alice signs message m:
Pick random k, compute $r = g^k$,
$s \equiv k^{-1}(r + \text{hash}(m)a) \bmod \ell$.
Signature is (r, s).

Anybody can verify signature:
Compute $r^s - g^r \cdot (g^a)^{\text{hash}(m)}$;
accept if 0.

Valid signatures get accepted

$$
\begin{aligned}
r^s &= g^{k \cdot k^{-1}(r+\mathsf{hash}(m)a)} \\
&= g^{r+\mathsf{hash}(m)a} \\
&= g^r \cdot (g^a)^{\mathsf{hash}(m)}.
\end{aligned}
$$

Thus difference is 0.

The discrete-logarithm problem

Let $p = 1000003$ and $g = 2$.
The number of elements in $\mathbf{F}_p^*$ is

The discrete-logarithm problem

Let $p = 1000003$ and $g = 2$.
The number of elements in $\mathbf{F}_p^*$ is
$1000002 = 2 \cdot 3 \cdot 166667$
and g has order 1000002.

The discrete-logarithm problem

Let $p = 1000003$ and $g = 2$.
The number of elements in $\mathbf{F}_p^*$ is
$1000002 = 2 \cdot 3 \cdot 166667$
and g has order 1000002.
In general, any element of $\mathbf{F}_p^*$ has order dividing $(p-1)$.
Here, $g = 2$ generates the entire multiplicative group modulo p.

Any $1 \leq h \leq p-1$ is power of g.
$h = 159429$, find n with $h = g^n$.
Could find n by brute force.
Is there a faster way?

<u>Understanding brute force</u>

Can compute successively
$g^1 = 2$,
$g^2 = 4$,
$g^3 = 8$,
$g^4 = 16$,
$\ldots$
$g^{20} = 48573$
$g^{1000001} = 500002 = g^{-1}$.
$g^{1000002} = 1$.

At some point we'll find n
with $g^n = 159429$.

Maximum cost of computation:
≤ 1000001 multiplications by g.

$\leq$ 1000001 nanoseconds on CPU that does 1 MULT/nanosecond. This is negligible work for $p \approx 2^{20}$.

But users can standardize a larger p, making the attack slower.

Attack cost scales linearly:
$\approx 2^{50}$ MULTs for $p \approx 2^{50}$,
$\approx 2^{100}$ MULTs for $p \approx 2^{100}$, etc.

(Not exactly linearly: cost of MULTs grows with p. But this is a minor effect.)

Computation has a good chance of finishing earlier.

Chance scales linearly:
1/2 chance of 1/2 cost;
1/10 chance of 1/10 cost; etc.

“So users should choose large n.”

Computation has a good chance of finishing earlier.
Chance scales linearly:
1/2 chance of 1/2 cost;
1/10 chance of 1/10 cost; etc.

"So users should choose large n."

That's pointless. We can apply "random self-reduction":
choose random r, say 69961;
compute $g^r = 872477$;
compute $g^{r+n} = g^r \cdot h$ as
$872477 \cdot 159429 = 718342$;
compute discrete log;
subtract r mod 1000002; get n.

Computation can be parallelized.

One low-cost chip can run
many parallel searches.
Example, 2^6 €: one chip,
2^{10} cores on the chip,
each 2^{30} MULTs/second?
Maybe; see SHARCS workshops
for detailed cost analyses.

Attacker can run
many parallel chips.
Example, 2^{30} €: 2^{24} chips,
so 2^{34} cores,
so 2^{64} MULTs/second,
so 2^{89} MULTs/year.

Multiple targets and giant steps

Computation can be applied to many targets at once.

Given 100 DL targets g^{n_1}, g^{n_2}, . . . , $g^{n_{100}}$:
Can find *all* of $n_1, n_2, \ldots, n_{100}$ with $\leq$ 1000002 MULTs.

Simplest approach: First build a sorted table containing g^{n_1}, . . . , $g^{n_{100}}$.
Then check table for g^1, g^2, etc.

Interesting consequence #1:
Solving all 100 DL problems
isn't much harder than
solving one DL problem.

Interesting consequence #2:
Solving *at least one*
out of 100 DL problems
is much easier than
solving one DL problem.

When did this computation
find its *first* n_i?

Interesting consequence #1:
Solving all 100 DL problems
isn't much harder than
solving one DL problem.

Interesting consequence #2:
Solving *at least one*
out of 100 DL problems
is much easier than
solving one DL problem.

When did this computation
find its *first* n_i?
Typically $\approx 1000002/100$ mults.

Can use random self-reduction
to turn a single target
into multiple targets.
Let ℓ be the order of g.

Given g^n:
Choose random $r_1, r_2, \ldots, r_{100}$.
Compute $g^{r_1} \cdot g^n$,
$g^{r_2} \cdot g^n$, etc.

Solve these 100 DL problems.
Typically $\approx \ell/100$ mults
to find *at least one*
$r_i + n \bmod \ell$,
immediately revealing n.

Also spent some MULTs
to compute each g^{r_i}:
$\approx \log_2 p$ MULTs for each i.

Faster: Choose $r_i = ir_1$
with $r_1 \approx \ell/100$.
Compute g^{r_1};
$g^{r_1} \cdot g^n$;
$g^{2r_1} \cdot g^n$;
$g^{3r_1} \cdot g^n$; etc.
Just 1 MULT for each new i.

$\approx 100 + \log_2 \ell + \ell/100$ MULTs
to find n given g^n.

Faster: Increase 100 to $\approx \sqrt{\ell}$.
Only $\approx 2\sqrt{\ell}$ MULTs
to solve one DL problem!
"Shanks baby-step-giant-step
discrete-logarithm algorithm."

Example: $p = 1000003$,
$\ell = 1000002$, $\sqrt{\ell} \approx 1000$.
$g = 2$, $h = g^n = 159429$.

Compute $g^{1000} = 510646$.
Then compute 1000 targets:
$h = g^0 \cdot g^n = 159429$,
$g^{1000} \cdot g^n = 536901$,
$g^{2 \cdot 1000} \cdot g^n = 525551$,
$g^{3 \cdot 1000} \cdot g^n = 710839$,
$g^{4 \cdot 1000} \cdot g^n = 3036$,

...
$g^{999 \cdot 1000} \cdot g^n = 143529$,

Build a sorted table of targets:

$g^{4\cdot1000}\cdot h = 3036$,
$g^{486\cdot1000}\cdot h = 3973$,
$g^{648\cdot1000}\cdot h = 5038$,
$g^{909\cdot1000}\cdot h = 7814$,
$g^{544\cdot1000}\cdot h = 7862$,

. . .

$g^{100\cdot1000}\cdot h = 999018$,

Look up g, g^2, g^3, etc. in table.

$g^{675} = 913004$; find
$g^{590\cdot1000}\cdot h = 913004$
in the table of targets.

Thus

$$675 \equiv 590 \cdot 1000 + n \bmod 1000002;$$

and

$$\begin{aligned} n &\equiv -590 \cdot 1000 + 675 \\ &\equiv 410677 \bmod 1000002. \end{aligned}$$

Test: $g^{410677} = 159429$.

More common version:

Let $m = \left\lfloor \sqrt{\ell} \right\rfloor$.

Compute table with (g^i, i)

for $0 \le i < m$;

sort while computing.

Each step costs 1 MULT.

Reach g^m, invert: $G = g^{-m}$.

Compute $G^j h$ and

compare with table entries.

Match instantly gives

$g^{-jm} h = g^i$, thus $n = i + jm$.

Cost: $(\le 2m + 2)$ MULTs $+1$INV.

Rationale

Write $n = n_0 + n_1 m$.
Then the baby step g^{n_0} matches the giant step $G^{n_1} h = g^{-n_1 m} h$.

Optimizations

Using $g^{jm} h$ avoids inversion but needs reduction mod $p - 1$ (extra implementation).

Can optimize by interleaving baby and giant steps (needs $\log_2 n$ MULTs for exponentiation again).