# The equivalence of the computational Diffie–Hellman and discrete logarithm problems in certain groups

David Fifield

January 7, 2012

**Abstract**

Whether the discrete logarithm problem can be reduced to the Diffie–Hellman problem is a celebrated open question. The security of Diffie–Hellman key exchange and other cryptographic protocols rests on the assumed difficulty of the computational Diffie–Hellman problem; such a reduction would show that this is equivalent to assuming that computing discrete logarithms is hard.

What is known is that a near-reduction exists for general groups, assuming that a conjecture about the existence of smooth numbers in an interval is true. Given access to a Diffie–Hellman oracle, and a small amount of additional information (this being the parameters of certain elliptic curves with smooth order), it is possible to compute discrete logarithms using a polylogarithmic number of calls to the oracle.

## 1 Introduction

It is well known that, computational Diffie–Hellman hardness assumption is at least as strong as the discrete logarithm hardness assumption, but whether the assumptions are equivalent is an open question. It would be nice to show that solving the Diffie–Hellman problem is at least as hard as computing discrete logarithms, both to eliminate one of the cryptographic hardness assumptions, and for the insight that such an equivalence might give into the structure of groups. Ideally, we want to show that there is no shortcut for solving Diffie–Hellman that is meaningfully faster than doing the straightforward discrete log calculation.

The first steps toward finding an equivalence are due to den Boer [4], who proved equivalence in groups $\mathbb{Z}_p^\times$ where $p$ is a prime such that the Euler totient of $p-1$ is smooth. A key idea in this first reduction is that of computing an answer modulo small primes, and then combining them using the Chinese remainder theorem. Maurer [6] gave an algorithm that proved equivalence when a small amount of extra information depending on the group order is provided. The extra information is the parameters of an elliptic curve of smooth order, for finding which no efficient algorithm is known. Maurer's algorithm is a generalization of den Boer's in a way similar to how ECM factoring [5] is a generalization of the $(p-1)$ factoring method [9]. Muzereau et al. [8] computed curve parameters for curves used in real cryptographic applications, for example including the NIST recommended curves, and gave a careful analysis of the complexity of the algorithm. Bentahar [1] tightened the reduction with the goal of minimizing the number of calls to an oracle, to bring the hardness of Diffie–Hellman and discrete logarithms as close as possible to one another. The thrust of the current paper is an exposition of Maurer's algorithm.

## 2 Definitions

Let $G$ be a finite cyclic group generated by a group element $g$. We will use $p$ to denote any divisor of $|G|$, such that $|G| = p \cdot h$. (Often $p = |G|$ and $h = 1$.) The *discrete logarithm problem* in the group $G$ is as follows: Given $g$ and $x \in G$, find an integer $a$ such that $g^a = x$. We will assume that the order of $G$ is known and so $a$ is unique within $[0, |G|)$. With $a$ restricted in this way, the discrete logarithm problem can be written in an easier-to-remember way: given $g$ and $g^a$, find $a$.

Let $G$ and $g$ be as above. The *computational Diffie–Hellman problem* is as follows: Given $g$ and two other group elements $x = g^a$ and $y = g^b$, compute $g^{ab}$. A related problem, the *decisional Diffie–Hellman problem*, only requires distinguishing whether a given $z$ is equal to $g^{ab}$ or not. In this paper we are concerned only with the computational version of the problem, and so will refer to it as *the* Diffie–Hellman problem.

A Diffie–Hellman oracle (DH oracle) for a group $G$ generated by $g$ is a notional black box that solves an instance of the Diffie–Hellman problem in one step. That is, given inputs $x = g^a$ and $y = g^b$, it outputs $g^{ab}$. We will denote this operation by $\mathrm{DH}(g^a, g^b) = g^{ab}$.

The notation $E_{A,B}(\mathbb{F}_p)$ stands for the elliptic curve $y^2 = x^3 + Ax + B$, where $x$, $y$, $A$, and $B$ lie in $\mathbb{F}_p$. $\#E_{A,B}(\mathbb{F}_p)$ is the order of the corresponding group.

An integer is called $S$-*smooth* if it has no prime factors greater than $S$. We will often call a number simply "smooth" without naming a specific $S$.

Both the discrete logarithm and Diffie–Hellman problems are assumed to be difficult, and real-world cryptosystems rely on their hardness. No known algorithm solves either one efficiently. When the order of $G$ is known, the discrete logarithm problem in $G$ can be solved in time $O(\sqrt{|G|})$ using the baby step–giant step or Pollard rho methods. It is clear that solving Diffie–Hellman is no harder than computing discrete logs: Given $g$, $x$, and $y$, take the discrete log of $x = g^a$ to recover $a$, then output $y^a = (g^b)^a = g^{ab}$. This paper shows under what conditions the reverse situation holds: With an efficient way to solve the Diffie–Hellman problem (the DH oracle), it is possible to compute discrete logarithms. Another way to state this is, that there is no materially faster way to solve Diffie–Hellman than by computing discrete logarithms.

## 3 Implicit representation: Algebra in the exponent

One of the main ideas behind the reduction is that of the *implicit representation* of integers, which using a group element $g^a$ to stand for the integer $a$. We can add two implicitly represented numbers by multiplying their respective group elements: $g^a g^b = g^{a+b}$. That is, we can add $a$ and $b$ without knowing them explicitly, however the result we receive is also not known explicitly. Once granted a DH oracle, we can also implicitly multiply, divide and in fact implicitly compute any rational function with integer coefficients [6].

Following a convention of Boneh and Lipton [2], we will use square brackets to signify implicit representation. Let $a$ and $b$ be integers in the range $[0, |G|)$.

$$[a] := g^a \text{ as an element of } G$$
$$[b] := g^b \text{ as an element of } G$$

Implicit negation is done by inverting a group element within the group.

$$[-a] := g^{-a} = (g^a)^{-1}$$

As was shown above, implicit addition is group multiplication.

$$[a+b] := g^{a+b} = g^a g^b$$
$$[a-b] := g^{a-b} = g^a g^{-b}$$

Implicit multiplication uses one call to the DH oracle.

$$[ab] := g^{ab} = \mathrm{DH}(g^a, g^b)$$

Exponentiation can be done efficiently through repeated multiplication and squaring using $O(\log |G|)$ calls to the DH oracle. The multiplicative inverse of an integer $a$ is found by raising $a$ to the power $p-2$ in the exponent, because $a^{p-1} \equiv 1 \mod p$.

$$[a^{-1}] := g^{(a^{-1})} = g^{(a^{p-2})}$$

Testing the equality of two implicitly represented integers is the same as testing the equality of their implicit representations.

$$a = b \quad \Leftrightarrow \quad g^a = g^b \quad \Leftrightarrow \quad [a] = [b]$$

We will also have to test equality mod $p$, a divisor of $|G|$. If $|G| = p \cdot h$, then

$$a \equiv b \mod p \quad \Leftrightarrow \quad ha \equiv hb \mod |G| \quad \Leftrightarrow \quad [ha] = [hb].$$

It follows from the above that we can perform any algorithm on an implicit representation that uses only the above algebraic operations. In particular, there are modular square root algorithms like Tonelli–Shanks that are entirely algebraic. The algorithm in Section 5 needs to calculate square roots.

We will have occasion to use the implicit representation of points on an elliptic curve. If $P = (a, b)$ is a point on a curve, then

$$[P] := ([a], [b]) = (g^a, g^b).$$

The formulas for the group law on elliptic curves are algebraic, so it is well defined to add two points in implicit representation.

With this notation, the discrete log problem may be restated as follows. Given $[a]$, find $a$; or in other words, given the implicit representation of $a$, find its explicit representation.

## 4 Existence of elliptic curves with smooth order

The other key idea is the use auxiliary elliptic curves of smooth order. While den Boer's proof on works on groups that are "inherently smooth" in a sense, Maurer's works for any group for which a "smooth" curve can be found. The smallness of the factors of the curve's order make brute-force discrete logarithm calculations feasible on subproblems.

A natural question is whether such curves exist. It is known that the order of an elliptic curve $E_{A,B}(\mathbb{F}_p)$ must lie in the Hasse interval

$$p - 2\sqrt{p} + 1 \leq \#E_{A,B}(\mathbb{F}_p) \leq p + 2\sqrt{p} + 1$$

and that all orders in this interval are achievable for some values of $A$ and $B$. According to Canfield et al. ([3], restated in [6]),

$$\psi(n, n^{1/u})/n = u^{-(1+o(u))u}$$

for every fixed $u$. $\psi(n, y)$ is the number of integers less than or equal to $n$ that are $(y + 1)$-smooth. This shows that smooth numbers are not uncommon in the interval $[0, n]$. It is conjectured that this is also true on the Hasse interval, and our proof will rely on the truth of this conjecture.

Leaving aside the question of whether such curves exist, we turn to whether they can be found. In general, no efficient method is known. In small cases, one can try random elliptic curves until finding one with smooth order. If $p + 1$ is smooth, then supersingular curves (which necessarily have order $p + 1$) will work: If $p \equiv 3 \mod 4$, then the curves $y^2 = x^3 + Ax$ are supersingular; also if $p \equiv 2 \mod 3$, then the curves $y^2 = x^3 + B$ are supersingular. The construction of elliptic curves having complex multiplication allows for building different orders; this method was used by Muzereau et al. to find parameters for elliptic curve groups used in practice. [8]

## 5 The Maurer reduction

This theorem is taken directly from Maurer [6].

**Theorem 1** *Let $G = \langle g \rangle$ be an arbitrary cyclic group with order $|G| = \prod_{i=1}^{r} p_i^{e_i}$. If for each prime $p_i$ the parameters $A_i$ and $B_i$ of a cyclic elliptic curve $E_{A_i,B_i}(F_{p_i})$ with smooth order for a smoothness bound $S$ are given, then discrete logarithms in $G$ can be computed using $O(\log^2 |g|)$ calls to the DH oracle and $O((S/\log S) \log^2 |G|)$ group operations. If $e_i > 1$ for some $i$, then a DH oracle for subgroups of $G$ is also required.*

The proof of this theorem includes a number of technical details to handle special cases. We will first give a proof using some simplifying assumptions that forbid these special cases. Later we will show how they may be removed. We assume:

1. $|G|$ is equal to a prime $p$. $A$ and $B$ denote the parameters of the cyclic elliptic curve over $\mathbb{F}_p$ with smooth order corresponding to this $p$.

2. At each step where it is required, a discrete logarithm is the $x$-coordinate of some point on the elliptic curve $E_{A_i,B_i}(\mathbb{F}_p)$. In other words, if we are solving the subproblem $g^a$ and seek $a$, then $a^3 + A_i a + B_i$ is a quadratic residue mod $p$: there is a $b$ such that $b^2 = a^3 + A_i a + B_i$.

Other than these assumptions, let conditions be as stated in the theorem. We will show how to use the DH oracle to compute discrete logarithms in $G$. Suppose we are given $x = [a] = g^a \in G$ and are asked to find $a$.

From $[a]$ we compute $[z] = [a^3 + Aa + B]$. We then take the square root of this quantity $[b]$ so that $[b^2] = [a^3 + Aa + B]$; this implies that the point $(a, b)$ lies on $E_{A,B}(\mathbb{F}_p)$. (Here we use assumption 2 that the square root exists.) We can do these calculations using the techniques of Section 3.

The point $Q = (a, b)$ encodes the answer to the discrete logarithm problem in its $x$-coordinate, however we know only its implicit representation $[Q] = ([a], [b])$. Let $P$ be a generator of $E_{A,B}(\mathbb{F}_p)$. If we can find $k$, the discrete logarithm of $[Q]$ with respect to $[P]$, then we know that $[Q] = k[P]$ and also that $Q = kP$. We may then recover $a$ as the $x$-coordinate of $kP$.

Let the factorization of $\#E_{A,B}(\mathbb{F}_p)$ be $q_1^{f_1} q_2^{f_2} \cdots q_r^{f_r}$. We will calculate the discrete logarithm $k_i$ modulo each of the factors $q_i$. Let

$$[V_i] = \frac{|E|}{q_i}[Q]$$

and then let $j$ range from 0 to $q_i - 1$ to compute the sequence

$$[P_{ij}] = j\frac{|E|}{q_i}[P]$$

until $[V_i] = [P_{ij}]$. (This is a brute-force discrete logarithm calculation of $\frac{|E|}{q_i}[Q]$ over the base $\frac{|E|}{q_i}[P]$. It is feasible because the elliptic curve order is smooth and therefore each $q_i$ is small.) Because $[V_i]$ has order $q_i$, $[V_i] = [P_{ij}]$ if and only if $j \equiv k \mod q_i$. Doing this for all the prime factors gives $k$ in modular representation, from which it can be recovered by the Chinese remainder theorem.

Let us now remove assumption 1. The algorithm above allows us to compute $k$ not only when $|G| = p$, but also $k_\ell$ for any $p$ that divides $|G|$. If all the factors of $|G|$ are distinct, we can compute a modular representation of the overall $k$ just by repeating the algorithm for each divisor. In the case that of a multiple prime factor of $|G|$, say $p^e \backslash |G|$, then we must calculate $k_\ell$ modulo $p^e$ and not $p$. One way to do this is with a DH oracle on a subgroup of $G$; details are given by Maurer and Wolf [7].

For assumption 2, if $a^3 + A_i a + B_i$ is not a quadratic residue mod $p$, then choose random $d \in \mathbb{F}_p$ and set $a' := a + d$ until $(a')^3 + A_i(a') + B_i$ is a quadratic residue. An expected number of only two choices of $d$ is required before finding a residue. Now when $(a', b)$ is recovered as a point on $E_{A_i,B_i}(\mathbb{F}_p)$, $a = a' - d$ can be computed.

We omit a discussion of the algorithm's complexity. Muzereau et al. have done a careful analysis [8].

## 6 Worked example

We will now work through an example, start to finish, of computing a discrete logarithm using a DH oracle. The base group will be small enough that we can simulate the DH oracle by taking a discrete log, however we will treat the oracle as a black box and not use the fact that it is internally calculating logarithms. The Sage [10] source code used to do the calculations in this section is included in Appendix A.

The discrete logarithm instance is as follows. Computations are done in $\mathbb{F}_{227}^\times$; $G$ is the order-113 subgroup generated by 4. We are asked to take the discrete logarithm of 63.

$$g = 4 \in \mathbb{F}_{227}$$
$$x = g^a = 63$$

The solution to the problem is $a = 10$, because $4^{10} \equiv 63 \mod 227$, but we don't know that yet. $g$ has order $p = 113$.

The first step is to find an elliptic curve over $\mathbb{F}_p$ with smooth order. By making an appeal to Heaven, we find that the curve $E$

$$y^2 = x^3 + 48x + 4$$

has order $126 = 2 \cdot 3^2 \cdot 7$. (In fact this instance is small enough that the author tried random curves until finding a smooth one.) A generator of $E$ is $P = (16, 3)$. We will use $A = 48, B = 4$ in the equations below.

We calculate

$$[z] = [a^3 + Aa + B] = 44.$$

(Recall that square brackets indicate implicit representation: $[a^3 + Aa + B] := g^{a^3+Aa+B}$.) To see whether $z$ is a quadratic residue mod $p$, we test the equality

$$\begin{aligned} [z^{(p-1)/2}] &= [1] \\ g^{44^{56}} &= g \\ 4 &= 4. \end{aligned}$$

Since the equality holds, $z$ is a quadratic residue and we may proceed. (If $z$ had not been a residue, we would have repeated with $a + d$ for random $d$ until it was.)

We now seek the implicit representation of a point on $E$ whose $x$-coordinate is $a$. This we do with a modular square root algorithm.

$$[b] = [z^{1/2}] = 27.$$

At this point we know that $(a, b)$ lies on $E$, however we do not know $(a, b)$, only its implicit representation $[Q] = ([a], [b]) = (63, 27)$.

The next step is to calculate the discrete logarithms (in the elliptic curve group) of the factors of the curve's order 2, $3^2$, and 7. We convert $P$ into its implicit representation $[P] = ([16], [3]) = (g^{16}, g^3) = (176, 64)$. Let $T = 126$ be the group order. For each $q$ dividing $T$, we compute

$$\frac{T}{q}[Q] \quad \text{and} \quad i \cdot \frac{T}{q}[P]$$

for various small values of $i$ until finding equality. The results are

$$\begin{aligned} \frac{T}{2}[Q] &= 1 \cdot \frac{T}{2}[P] \\ \frac{T}{3}[Q] &= 0 \cdot \frac{T}{3}[P] \\ \frac{T}{7}[Q] &= 5 \cdot \frac{T}{7}[P] \end{aligned}$$

This leads to the congruences

$$\begin{aligned} k &\equiv 1 \mod 2 \\ k &\equiv 0 \mod 3 \\ k &\equiv 5 \mod 7 \end{aligned}$$

The Chinese remainder theorem gives the unique $k \in [0, T]$ that satisfies these congruences. We take $k$ times the elliptic curve generator $P$, and finally recover $a$ by taking the $x$-coordinate of $kP$.

$$\begin{aligned} k &= 33 \\ kP &= (10, 69) \\ a &= 10. \end{aligned}$$

## 7 Summary

In certain groups, solving the Diffie–Hellman problem is equivalent to computing discrete logarithms. Specifically, this is true in a cyclic group $G$ generated by $g$ if:

- The conjecture in Section 4 is true over the Hasse interval for each of the factors $p_i$ of $|G|$. This implies the existence of elliptic curves with smooth order over $\mathbb{F}_{p_i}$.

- The mentioned curves are already known or can be efficiently computed.

The import of this fact is that in groups where the above condition holds, there is no asymptotically faster way to solve the computational Diffie–Hellman problem than by the obvious discrete logarithm–based method. Cryptosystems that base their security on the computational Diffie–Hellman assumption can, in some cases, rest on the discrete logarithm hardness assumption instead. Auxiliary elliptic curve parameters are known for groups used in practice.

## References

[1] K. Bentahar. The equivalence between the DHP and DLP for elliptic curves used in practical applications, revisited. In Nigel Smart, editor, *Cryptography and Coding*, Lecture Notes in Computer Science, pages 376–391. Springer Berlin / Heidelberg, 2005.

[2] D. Boneh and R. Lipton. Algorithms for black-box fields and their application to cryptography. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 283–297. Springer Berlin / Heidelberg, 1996.

[3] E. R. Canfield, Paul Erdős, and Carl Pomerance. On a problem of Oppenheim concerning "factorisatio numerorum". *Journal of Number Theory*, 17(1):1–28, 1983.

[4] B. den Boer. Diffie–Hellman is as strong as discrete log for certain primes. In Shafi Goldwasser, editor, *Advances in Cryptology – CRYPTO '88*, volume 403 of *Lecture Notes in Computer Science*, pages 530–539. Springer Berlin / Heidelberg, 1990.

[5] H. W. Lenstra Jr. Factoring integers with elliptic curves. *The Annals of Mathematics*, 126(3):649–673, 1987.

[6] U. Maurer. Towards the equivalence of breaking the Diffie–Hellman protocol and computing discrete logarithms. In Yvo Desmedt, editor, *Advances in Cryptology – CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 271–281. Springer Berlin / Heidelberg, 1994.

[7] U. Maurer and S. Wolf. Diffie–Hellman oracles. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 268–282. Springer Berlin / Heidelberg, 1996.

[8] A. Muzereau, N. P. Smart, and F. Vercauteren. The equivalence between the DHP and DLP for elliptic curves used in practical applications, 2004.

[9] J. M. Pollard. Theorems on factorization and primality testing. *Proceedings of the Cambridge Philosophical Society*, 76:521–528, 1974.

[10] W. A. Stein et al. *Sage Mathematics Software (Version 4.7.2)*. The Sage Development Team, 2011. http://www.sagemath.org.

## A Source code for worked example

This paper and source code can be downloaded from http://www.bamsoftware.com/stanford/cs259c/.

```
set_random_seed(1)

def DH_maker(g):
    """Return a DH oracle function for the given generator."""
    def f(ga, gb):
        a = discrete_log(ga, g)
        return gb^a
    return f
```

```
def is_smooth(n, S):
    return factor(n)[-1][0] <= S

def find_smooth_ec(R, S):
    """Return an elliptic curve whose order has no factors greater than S."""
    while True:
        try:
            A, B = R.random_element(), R.random_element()
            E = EllipticCurve(R, [A, B])
            if is_smooth(E.order(), S) and len(E.gens()) == 1:
                break
        except ArithmeticError:
            # Accidentally picked a singular curve.
            pass
    return E

def imp_add(ga, gb):
    return ga * gb

def imp_mul(ga, gb):
    return DH(ga, gb)

def imp_pow(g, ga, x):
    """Return [a^x] given [a] and x."""
    # [p] = 1
    gt = g
    while x != 0:
        if x % 2 == 1:
            # [t] = [t * a]
            gt = imp_mul(gt, ga)
        # [a] = [a * a]
        ga = imp_mul(ga, ga)
        x >>= 1
    return gt

def imp_inv(g, ga):
    p = g.multiplicative_order()
    return imp_pow(g, ga, p - 2)

def imp_is_quadratic_residue(g, ga, p):
    assert p.is_prime()
    h = Integer(g.multiplicative_order() / p)
    test = imp_pow(g, ga, Integer((p - 1) / 2))
    return test^h == g^h

def imp_sqrt(g, ga, p):
    """Calculate [x] such that [x^2] = [a] using the Tonelli--Shanks algorithm.
    The variables are named according to
    https://en.wikipedia.org/wiki/Tonelli%E2%80%93Shanks_algorithm."""
    assert p.is_prime()
    h = Integer(g.multiplicative_order() / p)
    assert imp_is_quadratic_residue(g, ga, p)
    if p % 4 == 3:
        return imp_pow(g, ga, Integer((p + 1) / 4))

    S = 0
    Q = p - 1
    while Q % 2 == 0:
        S += 1
        Q >>= 1
    assert p - 1 == Q * 2^S

    z = g^2
    while imp_is_quadratic_residue(g, z, p):
        z = imp_add(z, g)

    c = imp_pow(g, z, Q)
    R = imp_pow(g, ga, Integer((Q + 1) / 2))
    t = imp_pow(g, ga, Q)
    M = S
    while t^h != g^h:
        tt = t
        for i in range(1, M):
            tt = imp_mul(tt, tt)
            if tt^h == g^h:
                break
        else:
            assert False, (i, M)
        b = imp_pow(g, c, 2^(M - i - 1))
        R = imp_mul(R, b)
        t = imp_mul(t, imp_mul(b, b))
        c = imp_mul(b, b)
        M = i
    return R

INF = (object(), object())
def imp_ec_add(g, P1, P2, A, B):
    """Add two elliptic curve points given in implicit representation."""
    x1, y1 = P1
    x2, y2 = P2
    if P1 == INF:
        return P2
    elif P2 == INF:
        return P1
    elif x1 != x2:
        m = imp_mul(imp_add(y2, y1^-1), imp_inv(g, imp_add(x2, x1^-1)))
        x3 = imp_add(imp_add(imp_mul(m, m), x1^-1), x2^-1)
        y3 = imp_add(imp_mul(m, imp_add(x1, x3^-1)), y1^-1)
        return x3, y3
    elif y1 != y2:
        return INF
    elif y1 != 0:
        m = imp_mul(imp_add(imp_mul(x1, x1)^3, g^A), imp_inv(g, y1^2))
        x3 = imp_add(imp_mul(m, m), x1^-2)
        y3 = imp_add(imp_mul(m, imp_add(x1, x3^-1)), y1^-1)
        return x3, y3
    else:
        return INF

def imp_ec_mul(g, k, P, A, B):
    """Multiply an elliptic curve point given in implicit representation."""
    Q = INF
    while k != 0:
        if k % 2 == 1:
            Q = imp_ec_add(g, Q, P, A, B)
        P = imp_ec_add(g, P, P, A, B)
        k >>= 1
    return Q

def imp_ec_discrete_log(g, iQ, iP, A, B):
    """Find k such that [k]iP = iQ in implicit representation."""
    acc = INF
    for k in range(g.multiplicative_order()):
        if acc == iQ:
            return k
        acc = imp_ec_add(g, acc, iP, A, B)
    assert False, k

G = GF(227)
g = G(4)
p = g.multiplicative_order()
h = 1
assert p.is_prime()
DH = DH_maker(g)

S = 11
E = find_smooth_ec(GF(p), S)
print E
print factor(E.order())
P = E.gen(0)
print P
A, B = E.a4(), E.a6()

a = randint(1, p)
x = ga = g^a

print "a =", a
print "x = [a] =", x

# Goal: given ga = [a], find a.

d = 0
while True:
    print

    gad = imp_add(ga, g^d)
    print "d = 0  [a + d] =", gad
    # Calculate [z] = [(a + d)^3 + A(a + d) + B] = [(a + d)((a + d)^2 + A) + B].
    gz = imp_add(imp_mul(gad, imp_add(imp_mul(gad, gad), g^A)), g^B)
    print "[z] =", gz
    print "[z^{(p - 1)/2}] =", imp_pow(g, gz, Integer((p - 1) / 2))
    if imp_is_quadratic_residue(g, gz, p):
        break
    # Random offset.
    d = randrange(1, p)

print

# Now gz is a quadratic residue.
print "[z] =", gz
gb = imp_sqrt(g, gz, p)
print "[b] =", gb

# Now (a + d, b) is a point on the elliptic curve E.

# Get the implicit representation of points P and Q on E.
iP = (g^P.xy()[0], g^P.xy()[1])
iQ = (gad, gb)

print "[P] =", iP
print "[Q] =", iQ
print

T = E.order()
crt_residues = []
crt_moduli = []
for q, f in factor(T):
    print "q", q, "f", f
    iPP = imp_ec_mul(g, Integer(T / q), iP, Integer(A), Integer(B))
    iQQ = imp_ec_mul(g, Integer(T / q), iQ, Integer(A), Integer(B))
    print "T/q * [P] =", iPP
    print "T/q * [Q] =", iQQ
    i = imp_ec_discrete_log(g, iQQ, iPP, Integer(A), Integer(B))
    print "i =", i
    crt_residues.append(i)
    crt_moduli.append(q)

print

k = CRT(crt_residues, crt_moduli)
print "k =", k
kP = k * P
print "kP =", kP
ad = kP.xy()[0]
a = ad - d
print "a =", a
print "g^a =", g^a
assert ga == g^a
```