Compressing RSA/Rabin keys

D. J. Bernstein

Thanks to:
University of Illinois at Chicago
NSF CCR–9983950
Alfred P. Sloan Foundation
American Institute of Mathematics

Public keys

Each user publishes a key $U \in \{2^{2047}, 2^{2047} + 1, \ldots, 2^{2048} - 1\}$.

User knows prime factors of $U$.
Hopefully attacker doesn’t.

RSA: also publish big exponent $e$; use primes allowing $e$th roots.

Rabin: always use exponent 2; use primes in $3 + 4\mathbb{Z}$.

Williams: $3 + 8\mathbb{Z}$ and $7 + 8\mathbb{Z}$.

Many subsequent variants; e.g., “RSA” using exponent 3, and “RSA” using exponent 65537.
Public keys

Each user publishes a key $U \in \{2^{2047}, 2^{2047} + 1, \ldots, 2^{2048} - 1\}$.

User knows prime factors of $U$.
Hopefully attacker doesn't.

RSA: also publish big exponent $e$; use primes allowing $e$th roots.
Rabin: always use exponent 2; use primes in $3 + 4\mathbb{Z}$.
Williams: $3 + 8\mathbb{Z}$ and $7 + 8\mathbb{Z}$.
Many subsequent variants; e.g., “RSA” using exponent 3, and “RSA” using exponent 65537.

The compression question
Can store $U$ in 2048 bits.
Can store $U_1, U_2, \ldots$ randomly accessible.
Can we use fewer bits?

Knee-jerk answer: “No! If you can’t afford 2048 bits, switch to 256-bit elliptic curves.
http://cr.yp.to/ecdh.html”
But elliptic-curve signatures have slow verification.
Want a better answer.
Public keys

Each user publishes a key $U \in \{2^{2047}, 2^{2047} + 1, \ldots, 2^{2048} - 1\}$.

User knows prime factors of $U$. Hopefully attacker doesn’t.

RSA: also publish big exponent $e$; use primes allowing $eth$ roots.
Rabin: always use exponent 2; use primes in $3 + 4\mathbb{Z}$.
Williams: $3 + 8\mathbb{Z}$ and $7 + 8\mathbb{Z}$.
Many subsequent variants; e.g., “RSA” using exponent 3, and “RSA” using exponent 65537.

The compression question

Can store $U$ in 2048 bits.
Can store $U_1, U_2, \ldots, U_n$, randomly accessible, in $2048n$ bits.

Can we use fewer bits?

Knee-jerk answer: “No! If you can’t afford $2048n$ bits, switch to 256-bit elliptic curves. 
http://cr.yp.to/ecdh.html”

But elliptic-curve signatures have slow verification.
Want a better answer.
Each user publishes a key $U \in \{2^{2047}, \ldots, 2^{2048} - 1\}$. 

User knows prime factors of $U$. 

Hopefully attacker doesn’t.

RSA: also publish big exponent $e$; use primes allowing $\sqrt[n]{\ldots}$ roots. 

Rabin: always use exponent 2; use primes in $3 + 4\mathbb{Z}$. 

Williams: $3 + 8\mathbb{Z}$ and $7 + 8\mathbb{Z}$. 

Many subsequent variants; e.g., “RSA” using exponent 3, and “RSA” using exponent 65537.

The compression question

Can store $U$ in 2048 bits. 
Can store $U_1, U_2, \ldots, U_n$, randomly accessible, in $2048n$ bits.

Can we use fewer bits?

Knee-jerk answer: “No! If you can’t afford $2048n$ bits, switch to 256-bit elliptic curves. http://cr.yp.to/ecdh.html”

But elliptic-curve signatures have slow verification.
Want a better answer.

Recognizing lower entropy

$U \in \{2^{2047}, \ldots, 2^{2048} - 1\}$, so $U$ has top bit 1.

Don’t store that bit.

With Rabin-Williams:

Don’t store bottom 3 bits.

Better: Users never generate divisible by 3, 5, 7, 11, so only 480 possibilities for $U \mod 9240$. Replace bottom 13 bits with 9-bit encoding of $U \mod 9240$. 

The compression question

Can store $U$ in 2048 bits.
Can store $U_1, U_2, \ldots, U_n$, randomly accessible, in $2048n$ bits.

Can we use fewer bits?

Knee-jerk answer: “No! If you can’t afford 2048$n$ bits, switch to 256-bit elliptic curves. http://cr.yp.to/ecdh.html”

But elliptic-curve signatures have slow verification.
Want a better answer.

Recognizing lower entropy

$U \in \{2^{2047}, \ldots, 2^{2048} - 1\}$ so $U$ has top bit 1.
Don’t store that bit.

With Rabin-Williams: $U \in 5 + 8\mathbb{Z}$. Don’t store bottom 3 bits.

Better: Users never generate $U$ divisible by 3, 5, 7, 11, so only 480 possibilities for $U$ mod 9240. Replace bottom 13 bits with 9-bit encoding of $U$ mod 9240.
Recognizing lower entropy

$U \in \{2^{2047}, \ldots, 2^{2048} - 1\}$
so $U$ has top bit 1.
Don’t store that bit.

With Rabin-Williams: $U \in 5 + 8\mathbb{Z}$.
Don’t store bottom 3 bits.

Better: Users never generate $U$ divisible by 3, 5, 7, 11,
so only 480 possibilities for $U$ mod 9240. Replace bottom 13 bits with 9-bit encoding of $U$ mod 9240.

Have reduced 2048 to 2043.
Can we do much better?

Knee-jerk answer: “No!
C’mon, you know you want to switch to elliptic curves.
http://cr.yp.to/ecdh.html”

But elliptic-curve signatures have slow verification.
Want a better answer.

E.g. User generates $p, q \in \{2^{1023}, \ldots, 2^{1024} - 1\},$
$q \in \{2^{1024}, \ldots, 2^{1025} - 1\},$
so $2 \log 2$ chance of $3, 5, 7, 11$, 
$\approx 1/1025 \log 2$ chance of $p$, 
$\approx 1/1026 \log 2$ chance of $q$, 
$\approx 1/8$ chance of $\{3, 5, 7, 11\}$,
so $> 2^{2023}$ equally.
Recognizing lower entropy

$U \in \{2^{2047}, \ldots, 2^{2048} - 1\}$
so $U$ has top bit 1.
Don’t store that bit.

With Rabin-Williams: $U \in 5 + 8\mathbb{Z}$.
Don’t store bottom 3 bits.

Better: Users never generate $U$
divisible by 3, 5, 7, 11,
so only 480 possibilities
for $U$ mod 9240. Replace
bottom 13 bits with 9-bit
encoding of $U$ mod 9240.

Have reduced 2048 to 2043.
Can we do much better?

Knee-jerk answer: “No!
C’mon, you know you want to
switch to elliptic curves.”

e.g. User generates $U = pq$ from
independent uniform random
$p \in \{2^{1023}, \ldots, 2^{1024} - 1\}$,
$q \in \{2^{1024}, \ldots, 2^{1025} - 1\}$:
$\approx 1/1025 \log 2$ chance of $p$ prime,
$\approx 1/1026 \log 2$ chance of $q$ prime,
$\approx 1/8$ chance of $\{3, 7\} + 8\mathbb{Z}$,
$\approx 2 \log 2 - 1$ chance of $pq < 2^{2048}$,
so $> 2^{2023}$ equally likely $U$’s.
Recognizing lower entropy

2047
2
2048
1
so
has top bit 1.
Don't store that bit.
With Rabin-Williams:
5 + 8
Z.
Don't store bottom 3 bits.
Better: Users never generate
divisible by 3
5
7
11,
so only 480 possibilities
for
mod 9240. Replace
bottom 13 bits with 9-bit
encoding of
mod 9240.

Have reduced 2048 to 2043.
Can we do much better?
Knee-jerk answer: "No!
C'mon, you know you want to
switch to elliptic curves."
e.g. User generates \( U = pq \) from independent uniform random
\( p \in \{2^{1023}, \ldots, 2^{1024} - 1\} \),
\( q \in \{2^{1024}, \ldots, 2^{1025} - 1\} \): 
\[
\approx 1/1025 \log 2 \text{ chance of } p \text{ prime,}
\approx 1/1026 \log 2 \text{ chance of } q \text{ prime,}
\approx 1/8 \text{ chance of } \{3, 7\} + 8\mathbb{Z},
\approx 2 \log 2 - 1 \text{ chance of } pq < 2^{2048},
\]
so > 2^{2023} equally likely \( U \)'s.

Reducing entropy

Define \( f(U) = 500 \text{th bit of } U \),
\( g(U) = U \) with 500th bit omitted.
Change key-generation procedure to produce keys \( U \) with
\( f(U) = 0 \).
Then can encode \( U \) with
saving one bit; also
top/bottom bits again.

Brute-force key generation:
generate \( U \) by the old method;
if \( f(U) = 1 \), try again.
Conjecturally this takes
almost exactly 2 tries on average;
confirmed by experiment.
Have reduced 2048 to 2043. Can we do much better? Knee-jerk answer: “No! C’mon, you know you want to switch to elliptic curves.”

e.g. User generates $U = pq$ from independent uniform random
$p \in \{2^{1023}, \ldots, 2^{1024} - 1\}$,
$q \in \{2^{1024}, \ldots, 2^{1025} - 1\}$:
$\approx 1/1025 \log 2$ chance of $p$ prime,
$\approx 1/1026 \log 2$ chance of $q$ prime,
$\approx 1/8$ chance of $\{3, 7\} + 8\mathbb{Z}$,
$\approx 2 \log 2 - 1$ chance of $pq < 2^{2048}$, so $> 2^{2023}$ equally likely $U$’s.

Reducing entropy
Define $f(U) = 500$th bit of $U$, $g(U) = U$ with 500th bit omitted.
Change key-generation procedure to produce keys $U$ with $f(U) = 0$. Then can encode $U$ as $g(U)$, saving one bit; also save top/bottom bits as before.

Brute-force key generation: generate $U$ by the old method; if $f(U) = 1$, try again. Conjecturally this takes almost exactly 2 tries on average; confirmed by experiment.
Reducing entropy

Define \( f(U) = 500th \) bit of \( U \),
\( g(U) = U \) with 500th bit omitted.

Change key-generation procedure to produce keys \( U \) with \( f(U) = 0 \).
Then can encode \( U \) as \( g(U) \), saving one bit; also save top/bottom bits as before.

Brute-force key generation:
generate \( U \) by the old method;
if \( f(U) = 1 \), try again.
Conjecturally this takes almost exactly 2 tries on average; confirmed by experiment.

More generally, select functions:
\( f : \{2048\text{-bit strings}\} \to \{k\text{-bit strings}\} \)
\( g : \{2048\text{-bit strings}\} \to \{(2048 - k)\text{-bit strings}\} \)
with \( f \times g \) invertible.

Change key-generation procedure to produce keys \( U \) with \( f(U) = 0 \).
Then can encode \( U \) as \( g(U) \), saving \( k \) bits.

Is \( f \times g \) easy to compute and easy to invert? Yes for the functions we'll consider.
Reducing entropy

Define $f(U) = 500$th bit of $U$, $g(U) = U$ with 500th bit omitted.

Change key-generation procedure to produce keys $U$ with $f(U) = 0$. Then can encode $U$ as $g(U)$, saving one bit; also save top/bottom bits as before.

Brute-force key generation: generate $U$ by the old method; if $f(U) = 1$, try again. Conjecturally this takes almost exactly 2 tries on average; confirmed by experiment.

More generally, select functions $f : \{2048$-bit strings$\} \rightarrow \{k$-bit strings$\}$ and $g : \{2048$-bit strings$\} \rightarrow \{(2048 - k)$-bit strings$\}$ with $f \times g$ invertible.

Change key-generation procedure to produce keys $U$ with $f(U) = 0$. Then can encode $U$ as $g(U)$, saving $k$ bits.

Is $f \times g$ easy to compute and easy to invert? Yes for the functions we’ll consider.
Reducing entropy

Define \( f(U) = 500 \text{th bit of } U \), \( g(U) = U \) with 500th bit omitted.

Change key-generation procedure to produce keys \( U \) with \( f(U) = 0 \).

Then can encode \( U \) as \( g(U) \), saving one bit; also save top/bottom bits as before.

Brute-force key generation: generate \( U \) by the old method; if \( f(U) = 1 \), try again.

Conjecturally this takes almost exactly 2 tries on average; confirmed by experiment.

More generally, select functions:

\[ f : \{2048 \text{-bit strings}\} \rightarrow \{k \text{-bit strings}\} \text{ and } \]
\[ g : \{2048 \text{-bit strings}\} \rightarrow \{(2048 - k) \text{-bit strings}\} \]
with \( f \times g \) invertible.

Change key-generation procedure to produce keys \( U \) with \( f(U) = 0 \).

Then can encode \( U \) as \( g(U) \), saving \( k \) bits.

Is \( f \times g \) easy to compute and easy to invert? Yes for the functions we’ll consider.

Do \( U \)'s exist with \( f(U) = 0 \)?

Conjecturally chance \( \frac{1}{2} \) for the functions we’ll consider.

(Provable for \( f \) chosen randomly from “universal” classes.)

Brute force takes \( 2^k \) tries; far too slow for large \( k \).

Can we do much better?

Yes. Will come back to this.

Are the resulting keys secure?

Not necessarily!
More generally, select functions
\[ f : \{2048\text{-bit strings}\} \to \{k\text{-bit strings}\} \text{ and} \]
\[ g : \{2048\text{-bit strings}\} \to \{(2048 - k)\text{-bit strings}\} \]
with \( f \times g \) invertible.

Change key-generation procedure to produce keys \( U \) with \( f(U) = 0 \).
Then can encode \( U \) as \( g(U) \), saving \( k \) bits.

Is \( f \times g \) easy to compute and easy to invert? Yes for the functions we’ll consider.

Do \( U \)'s exist with \( f(U) = 0 \)?
Conjecturally chance \( \approx 1/2^k \) for the functions we’ll consider.
(Provable for \( f \) chosen randomly from “universal” classes.)

Brute force takes \( \approx 2^k \) tries; far too slow for large \( k \).
Can we do much better? Yes. Will come back to this.

Are the resulting keys secure? Not necessarily!
Do $U$'s exist with $f(U) = 0$?
Conjecturally chance $\approx 1/2^k$ for the functions we'll consider.
(Provable for $f$ chosen randomly from “universal” classes.)

Brute force takes $\approx 2^k$ tries; far too slow for large $k$.
Can we do much better?
Yes. Will come back to this.

Are the resulting keys secure? Not necessarily!

The half-special number-field sieve
1998 Lenstra: “Numbers of the form $2^{1024} \pm t$ ... 1024-bit RSA security, as long as $t$ is not much smaller than $2^{500}$.
Chance of an “unusually small” NFS polynomial is "negligible."

Not true. Reducing entropy, using $f(U) = \text{half the bits of } U$ as $g(U)$, reduces conjectured security level.

Skewed NFS polynomials (1999 Murphy) turn out to be unusually small for these numbers.
Do $U$’s exist with $f(U) = 0$?
Conjecturally chance $\approx 1/2^k$
for the functions we’ll consider.
(Provable for $f$ chosen randomly from “universal” classes.)
Brute force takes $\approx 2^k$ tries;
far too slow for large $k$.
Can we do much better?
Yes. Will come back to this.
Are the resulting keys secure?
Not necessarily!

The half-special number-field sieve
1998 Lenstra: “Numbers of the form $2^{1024} \pm t \ldots$ offer regular
1024-bit RSA security, as long as $t$ is not much smaller than $2^{500}$.”
Chance of an “unusually small”
NFS polynomial is “negligible.”
Not true. Reducing entropy,
using $f(U) = \text{half the bits of } U$,
reduces conjectured security level.
Skewed NFS polynomials
(1999 Murphy) turn out to be
unusually small for these numbers.
Do $f(U) = 0$? Conjecturally chance $1/2^k$ for the functions we'll consider. (Provable for chosen randomly from "universal" classes.) 

$\approx 2^k$ tries; large $k$. Better?

Can we do much better? Yes. Will come back to this.

Are the resulting keys secure? Not necessarily!

The half-special number-field sieve

1998 Lenstra: “Numbers of the form $2^{1024} \pm t \ldots$ offer regular 1024-bit RSA security, as long as $t$ is not much smaller than $2^{500}$.”

Chance of an “unusually small” NFS polynomial is “negligible.”

Not true. Reducing entropy, using $f(U) = \text{half the bits of } U$, reduces conjectured security level.

Skewed NFS polynomials (1999 Murphy) turn out to be unusually small for these numbers.

Sharing entropy

Generate random $U_1$ from set $S$ of all possible keys.

Define $S_1 = S \cap f(U_1)$.

Generate random $U_2$ from $S_1$; e.g., for $f = 500$th bit, generate random $U_2$ having same 500th bit as $U_1$.

Similarly generate $U_3$.

Compress $U_2$ to $g(U_2)$; compress $U_3$ to $g(U_3)$; etc.

Overall $(2048 - k)$ bits to store $U_1, U_2, \ldots$. 

The half-special number-field sieve

1998 Lenstra: “Numbers of the form $2^{1024} \pm t \ldots$ offer regular 1024-bit RSA security, as long as $t$ is not much smaller than $2^{500}$.”

Chance of an “unusually small” NFS polynomial is “negligible.”

Not true. Reducing entropy, using $f(U) = \text{half the bits of } U$, reduces conjectured security level.

Skewed NFS polynomials (1999 Murphy) turn out to be unusually small for these numbers.

Sharing entropy

Generate random $U_1$
from set $S$ of all possible keys.
Define $S_1 = S \cap f^{-1}(f(U_1))$.

Generate random $U_2 \in S_1$
(e.g., for $f = 500$th bit,
generate random $U_2$ having same 500th bit as $U_1$).
Similarly generate $U_3, U_4, \ldots$.

Compress $U_2$ to $g(U_2)$;
compress $U_3$ to $g(U_3)$; etc.
Overall $(2048 - k)n + k$ bits to store $U_1, U_2, \ldots, U_n$. 
Sharing entropy

Generate random $U_1$
from set $S$ of all possible keys.
Define $S_1 = S \cap f^{-1}(f(U_1))$.

Generate random $U_2 \in S_1$:
e.g., for $f = 500$th bit,
generate random $U_2$ having
same 500th bit as $U_1$.
Similarly generate $U_3, U_4, \ldots$.

Compress $U_2$ to $g(U_2)$;
compress $U_3$ to $g(U_3)$; etc.
Overall $(2048 - k)n + k$ bits
to store $U_1, U_2, \ldots, U_n$.

If distribution of $U_1$ is uniform over $S$, and distribution of $U_2$ given $U_1$ is uniform over $S_1$, then distribution of $U_2$ is uniform over $S$.

So attacker's chance of factoring $U_2$ is provably identical to attacker's chance of factoring $U_1$.

Same comment with "factoring" replaced by "forging signatures" etc.

Sharing entropy is provably secure.
Sharing entropy

Generate random $U_1$ from set $S$ of all possible keys. Define $S_1 = S \cap f^{-1}(f(U_1))$.

Generate random $U_2 \in S_1$: e.g., for $f = 500$th bit, generate random $U_2$ having same 500th bit as $U_1$. Similarly generate $U_3, U_4, \ldots$.

Compress $U_2$ to $g(U_2)$; compress $U_3$ to $g(U_3)$; etc. Overall $(2048 - k)n + k$ bits to store $U_1, U_2, \ldots, U_n$.

If distribution of $U_1$ is uniform over $S$, and distribution of $U_2$ given $U_1$ is uniform over $S_1$, then distribution of $U_2$ is uniform over $S$.

So attacker’s chance of factoring $U_2$ is provably identical to attacker’s chance of factoring $U_1$. Same comment with “factoring” replaced by “forging signatures” etc.

Sharing entropy is provably secure.
If distribution of $U_1$ is uniform over $S$, and distribution of $U_2$ given $U_1$ is uniform over $S_1$, then distribution of $U_2$ is uniform over $S$.

So attacker’s chance of factoring $U_2$ is provably identical to attacker’s chance of factoring $U_1$. Same comment with “factoring” replaced by “forging signatures” etc.

Problem arises with or without shared entropy. (e.g., Coppersmith, Bernstein)

For safety, choose key sizes so that (conjecturally) attacker can’t even do one factorization.

Time to factor $U_1$ and $U_2$ can be less than double the time for a single factorization (e.g., Schnorr, Eratosthenes)

Analogy: brute-force search versus a secret-key cipher finds $n$ target keys as fast as finding one target key.

Sharing entropy is provably secure.
If distribution of $U_1$
is uniform over $S$,
and distribution of $U_2$ given $U_1$
is uniform over $S_1$,
then distribution of $U_2$
is uniform over $S$.

So attacker’s chance of factoring $U_2$
is provably identical to
attacker’s chance of factoring $U_1$.
Same comment with “factoring”
replaced by “forging signatures”
etc.

Sharing entropy is provably secure.

Time to factor $U_1$ and $U_2$
can be less than double
the time for a single factorization.
(e.g., Schnorr, Eratosthenes)

Analogy: brute-force search
versus a secret-key cipher
finds $n$ target keys in same time
as finding one target key.

Problem arises with or without
shared entropy.
(e.g., Coppersmith, Bernstein)

For safety, choose key sizes
so that (conjecturally) attacker
can’t even do one factorization.
If distribution of $U_1$ is uniform over $\mathcal{U}$, and distribution of $U_2$ given $U_1$ is uniform over $\mathcal{U}$, then distribution of $U_2$ is uniform over $\mathcal{U}$.

So attacker's chance of factoring $U_2$ is provably identical to attacker's chance of factoring $U_1$.

Same comment with "factoring" replaced by "forging signatures" etc.

Sharing entropy is provably secure.

Time to factor $U_1$ and $U_2$ can be less than double the time for a single factorization. (e.g., Schnorr, Eratosthenes)

Analogy: brute-force search versus a secret-key cipher finds $n$ target keys in same time as finding one target key.

Problem arises with or without shared entropy. (e.g., Coppersmith, Bernstein)

For safety, choose key sizes so that (conjecturally) attacker can’t even do one factorization.

Perhaps time to factor $U_1$ or $U_2$ is below time to factor $U_1$.

Analogy: brute-force search finds some target key after $\approx 1/n$ of the computation.

As before, problem arises with or without shared entropy.

For safety, multiply conjectured factorization success chance (e.g. ECM success chance) by $n$ before choosing key sizes.
Time to factor $U_1$ and $U_2$ can be less than double the time for a single factorization. (e.g., Schnorr, Eratosthenes)

Analogy: brute-force search versus a secret-key cipher finds $n$ target keys in same time as finding one target key.

Problem arises with or without shared entropy. (e.g., Coppersmith, Bernstein)

For safety, choose key sizes so that (conjecturally) attacker can’t even do one factorization.

Perhaps time to factor $U_1$ or $U_2$ is below time to factor $U_1$.

Analogy: brute-force search finds some target key out of $n$ after $\approx 1/n$ of the computation.

As before, problem arises with or without shared entropy.

For safety, multiply conjectured factorization success chance (e.g. ECM success chance) by $n$ before choosing key sizes.
Perhaps time to factor $U_1$ or $U_2$ is below time to factor $U_1$.

Analogy: brute-force search finds some target key out of $n$ after $\approx 1/n$ of the computation.

As before, problem arises with or without shared entropy.

For safety, multiply conjectured factorization success chance (e.g. ECM success chance) by $n$ before choosing key sizes.

Is this overkill?

Are there algorithms to factor $U_1$ or $U_2$ or ... on $U_1$ more quickly than factoring $U_1$?

For discrete logs, prove "no" by randomized self-reduction.

For factorization, no hope of proof without an extra $n$. Factorization literature needs to explicitly address multiple inputs.

Maybe we're oversimplifying by considering just one input.
Perhaps time to factor $U_1$ or $U_2$ is below time to factor $U_1$.

Analogy: brute-force search finds some target key out of $n$ after $\approx 1/n$ of the computation.

As before, problem arises with or without shared entropy.

For safety, multiply conjectured factorization success chance (e.g. ECM success chance) by $n$ before choosing key sizes.

Is this overkill?
Are there algorithms to factor $U_1$ or $U_2$ or $\ldots$ or $U_n$ more quickly than factoring $U_1$?

For discrete logs, prove “no” by randomized self-reduction.

For factorization, no hope of proof without an extra $n$.
Factorization literature needs to explicitly address multiple inputs. Maybe we’re oversimplifying by considering just one input.
Is this overkill?

Are there algorithms to factor $U_1$ or $U_2$ or \ldots or $U_n$ more quickly than factoring $U_1$?

For discrete logs, prove “no” by randomized self-reduction.

For factorization, no hope of proof without an extra $n$.

Factorization literature needs to explicitly address multiple inputs. Maybe we’re oversimplifying by considering just one input.

Generating $U$ given bottom half

Define $f(U) = U$ mod $2^{1024}$.

Reasonably fast generation of $U$ with $f(pq) = f(U)$.

Choose 1024-bit $p$.

$q = 2^{1024} + (p^{-1} f(U) - 1)$

If not both primes, try again.

If $pq > 2^{2048}$, try again.

Conjecturally $\approx 2^{17}$ tries on average.
Is this overkill?

Are there algorithms to factor $U_1$ or $U_2$ or ... or $U_n$ more quickly than factoring $U_1$?

For discrete logs, prove “no” by randomized self-reduction.

For factorization, no hope of proof without an extra $n$.

Factorization literature needs to explicitly address multiple inputs. Maybe we’re oversimplifying by considering just one input.

Generating $U$ given bottom half

Define $f(U) = U \mod 2^{1024}$.

Reasonably fast generation of $p, q$ with $f(pq) = f(U_1)$, given $f(U_1)$:

Choose 1024-bit $p$. Compute $q = 2^{1024} + (p^{-1} f(U_1) \mod 2^{1024})$. If not both primes, try again.

If $pq > 2^{2048}$, try again.

Conjecturally $\approx 2^{17}$ tries on average.
Is this overkill? Are there algorithms to factor 1 or 2 or more quickly than factoring 1? For discrete logs, prove “no” by randomized self-reduction. For factorization, no hope of proof without an extra. Factorization literature needs to explicitly address multiple inputs. Maybe we're oversimplifying by considering just one input.

Generating $U$ given bottom half

Define $f(U) = U \mod 2^{1024}$.

Reasonably fast generation of $p, q$ with $f(pq) = f(U_1)$, given $f(U_1)$:
Choose 1024-bit $p$. Compute $q = 2^{1024} + (p^{-1} f(U_1) \mod 2^{1024})$.
If not both primes, try again.
If $pq > 2^{2048}$, try again.

Conjecturally $\approx 2^{17}$ tries on average.

Analogous method works for $f(U) = \lceil U/2^{1024} \rceil$.

Method reinvented several times. Published 1991 Guillou Quisquater, in context of reducing entropy: “Some forms of the modulus need less storage... all of the bits of the most significant byte valued to zero.”
Generating $U$ given bottom half

Define $f(U) = U \mod 2^{1024}$.

Reasonably fast generation of $p, q$ with $f(pq) = f(U_1)$, given $f(U_1)$:
Choose 1024-bit $p$. Compute $q = 2^{1024} + (p^{-1}f(U_1) \mod 2^{1024})$.
If not both primes, try again.
If $pq > 2^{2048}$, try again.

Conjecturally $\approx 2^{17}$ tries on average.

Analogous method works for $f(U) = \lfloor U/2^{1024} \rfloor$.

Method reinvented several times.

Published 1991 Guillou Quisquater, in context of reducing entropy:
“Some forms of the modulus ... need less storage. ... all of the bits of the $y$ most significant bytes are valued to zero.”
Analogous method works for \( f(U) = \lfloor U/2^{1024} \rfloor \).

Method reinvented several times.

Published 1991 Guillou Quisquater, in context of reducing entropy:

“Some forms of the modulus need less storage. . . . all of the bits of the \( y \) most significant bytes are valued to zero.”

Patent application filed 1995 by Vanstone and Zuccherato:

“A method of encrypting data selecting said public key having a plurality of sets of bits, at least one set being of a predetermined pattern of bits and applying said public key to encrypt the message.”

Includes some generation methods, ranging from sensible to silly.

Analogous method works for
\[ f(U) = \left\lfloor U/2^{1024} \right\rfloor. \]
Method reinvented several times.

Published 1991 Guillou Quisquater, in context of reducing entropy:
“Some forms of the modulus . . . need less storage. . . .
all of the bits of the \( y \) most significant bytes are valued to zero.”

Patent application filed 1995 by Vanstone and Zuccherato:
“A method of encrypting data . . . selecting said public key . . .
having a plurality of sets of bits, at least one set being of a predetermined pattern of bits . . .
and applying said public key to encrypt the message.”
Includes some generation methods, ranging from sensible to silly.

Analogous method works for $2^{1024}$.

Method reinvented several times. Published 1991 Guillou Quisquater, in context of reducing entropy: 

"Some forms of the modulus need less storage.

all of the bits of the most significant bytes are valued to zero."

Patent application filed 1995 by Vanstone and Zuccherato:

“A method of encrypting data... selecting said public key... having a plurality of sets of bits, at least one set being of a predetermined pattern of bits... and applying said public key to encrypt the message."

Includes some generation methods, ranging from sensible to silly.


More patents filed by Lenstra, responding to silly methods.

“Select a number; obtain the factor as $n'/p$; check whether the factor $q$ is prime; if the factor $q$ is prime, compute $n$ as the product of $p$ and $q$; determine that the number is the RSA modulus; and if the factor $q$ is not prime, adjust and repeat the check of whether the factor is prime.”

Patent application filed 1995 by Vanstone and Zuccherato:
“A method of encrypting data... selecting said public key...
having a plurality of sets of bits, at least one set being of a
predetermined pattern of bits... and applying said public key
to encrypt the message.”

Includes some generation methods, ranging from sensible to silly.


More patents filed by Lenstra, responding to silly methods.
“Select a number $p$; ... obtain the factor $q$ as $n'/p$; check whether the factor $q$ is prime; if the factor $q$ is prime, compute the number $n$ as the product of $p$ and $q$ and determine that the number $n$ is the RSA modulus; and if the factor $q$ is not prime, adjust $q$ and repeat the check of whether the factor $q$ is prime.”

More patents filed by Lenstra, responding to silly methods. “Select a number \( p \); \ldots \) obtain the factor \( q \) as \( n' / p \); check whether the factor \( q \) is prime; if the factor \( q \) is prime, compute the number \( n \) as the product of \( p \) and \( q \) and determine that the number \( n \) is the RSA modulus; and if the factor \( q \) is not prime, adjust \( q \) and repeat the check of whether the factor \( q \) is prime.”


These key-generation methods allow compression from 2048 bits to 1024 bits.

Exactly how fast is this? Can we make it even faster?

What if \( f(U) = U_{1280} \)?

What if \( f(U) = U_{1536} \)?

Do we still have fast key-generation methods?
More patents filed by Lenstra, responding to silly methods.

“Select a number $p$; ... obtain the factor $q$ as $n'/p$; check whether the factor $q$ is prime; if the factor $q$ is prime, compute the number $n$ as the product of $p$ and $q$ and determine that the number $n$ is the RSA modulus; and if the factor $q$ is not prime, adjust $q$ and repeat the check of whether the factor $q$ is prime.”


These key-generation methods allow compression from 2048 bits to 1024 bits.

Exactly how fast is this? Can we make it even faster?

What if $f(U) = U \mod 2^{1280}$?

What if $f(U) = U \mod 2^{1536}$?

Do we still have fast key-generation methods?
These key-generation methods allow compression from 2048 bits to 1024 bits.

Exactly how fast is this? Can we make it even faster?

What if \( f(U) = U \mod 2^{1280} \)?

What if \( f(U) = U \mod 2^{1536} \)?

Do we still have fast key-generation methods?

Unbalanced primes

Take \( f(U) = U \mod 2^{1280} \).

Choose 768-bit \( p \). Compute \( q = 2^{1280} + (p^{-1} f(U) \mod 2^{1280}) \).

If not both primes, try again. If \( pq > 2^{2048} \), try again.

This allows compression from 2048 bits to 768 bits with unbalanced \( p \).

(1998 Lenstra)

ECM more dangerous than NFS! Don’t want \( p \) so small.
These key-generation methods allow compression from 2048 bits to 1024 bits.

Exactly how fast is this? Can we make it even faster?

What if $f(U) = U \mod 2^{1280}$?
What if $f(U) = U \mod 2^{1536}$? Do we still have fast key-generation methods?

Unbalanced primes

Take $f(U) = U \mod 2^{1280}$.

Choose 768-bit $p$. Compute
$q = 2^{1280} + (p^{-1} f(U_1) \mod 2^{1280})$.
If not both primes, try again.
If $pq > 2^{2048}$, try again.

This allows compression from 2048 bits to 768 bits, with unbalanced $p, q$.

(1998 Lenstra)

ECM more dangerous than NFS! Don’t want $p$ so small.
These key-generation methods allow compression from 2048 bits to 1024 bits. Exactly how fast is this? Can we make it even faster? What if \( f(U) = U \mod 2^{1280} \)? What if \( f(U) = U \mod 2^{1536} \)? Do we still have fast key-generation methods?

Unbalanced primes

Take \( f(U) = U \mod 2^{1280} \).

Choose 768-bit \( p \). Compute \( q = 2^{1280} + (p^{-1} f(U_1) \mod 2^{1280}) \).

If not both primes, try again.
If \( pq > 2^{2048} \), try again.

This allows compression from 2048 bits to 768 bits, with unbalanced \( p, q \).
(1998 Lenstra)

ECM more dangerous than NFS!

Don’t want \( p \) so small.

Primes in lattices

Take \( f(U) = U \mod 2^{1366} \).

Choose 683-bit \( p_0 \). Compute \( q_0 = p_0^{-1} f(U_1) \mod 2^{1366} \).

Idea: will take \( p = p_0 + 2^{683} \)
and \( q = q_0 + 2^{683} \).

Use lattice reduction to try to find \( p_1, q_1 \)
with \( (f(U_1) - p_0 q_0) = p_1 q_0 + q_1 p_0 \) (mod \( 2^{683} \)).

Good chance of success.
(2003 Coppersmith)
**Unbalanced primes**

Take \( f(U) = U \mod 2^{1280} \).

Choose 768-bit \( p \). Compute \( q = 2^{1280} + (p^{-1} f(U_1) \mod 2^{1280}) \).

If not both primes, try again.

If \( pq > 2^{2048} \), try again.

This allows compression from 2048 bits to 768 bits, with unbalanced \( p, q \).

(1998 Lenstra)

ECM more dangerous than NFS!

Don't want \( p \) so small.

---

**Primes in lattices**

Take \( f(U) = U \mod 2^{1366} \).

Choose 683-bit \( p_0 \). Compute \( q_0 = p_0^{-1} f(U_1) \mod 2^{683} \).

Idea: will take \( p = p_0 + 2^{683} p_1 \) and \( q = q_0 + 2^{683} q_1 \).

Use lattice reduction to try to find \( p_1, q_1 \approx 2^{341} \) with \( \left( f(U_1) - p_0q_0 \right) / 2^{683} \equiv p_1q_0 + q_1p_0 \pmod{2^{683}} \).

Good chance of success.

(2003 Coppersmith)
Primes in lattices
Take $f(U) = U \mod 2^{1366}$.

Choose 683-bit $p_0$. Compute
$q_0 = p_0^{-1} f(U_1) \mod 2^{683}$.

Idea: will take $p = p_0 + 2^{683} p_1$
and $q = q_0 + 2^{683} q_1$.

Use lattice reduction
to try to find $p_1, q_1 \approx 2^{341}$
with $(f(U_1) - p_0 q_0)/2^{683} \equiv p_1 q_0 + q_1 p_0 \pmod{2^{683}}$.

Good chance of success.
(2003 Coppersmith)

This allows compression from 2048 bits to 682 bits,
with balanced $p, q$.

Minor flaw: uniform random
$p_0$ does not produce exactly
uniform random integer.
But confirm experimentally
that each $p_0$ has good chance
of producing at least one
$q_1$.
This implies that each choice
of $p$ has probability
not far above uniform.
Primes in lattices

Take $f(U) = U \mod 2^{1366}$.

Choose 683-bit $p_0$. Compute $q_0 = p_0^{-1} f(U_1) \mod 2^{683}$.

Idea: will take $p = p_0 + 2^{683} p_1$ and $q = q_0 + 2^{683} q_1$.

Use lattice reduction to try to find $p_1, q_1 \approx 2^{341}$ with $(f(U_1) - p_0 q_0) / 2^{683} \equiv p_1 q_0 + q_1 p_0 \pmod{2^{683}}$.

Good chance of success.

(2003 Coppersmith)

This allows compression from 2048 bits to 682 bits, with balanced $p, q$.

Minor flaw: uniform random $p_0$ does not produce exactly uniform random integer $p$.
But confirm experimentally that each $p_0$ has good chance of producing at least one $p$.
This implies that each choice of $p$ has probability not far above uniform.
This allows compression from 2048 bits to 682 bits, with balanced $p, q$.

Minor flaw: uniform random $p_0$ does not produce exactly uniform random integer $p$. But confirm experimentally that each $p_0$ has good chance of producing at least one $p$. This implies that each choice of $p$ has probability not far above uniform.

Some open questions:

Find random $p, q \approx 2^{1024}$ given $pq \mod 2^{1500}$? Maybe use higher-dimensional lattices.

Or $p \approx 2^{768}, q \approx 2^{1280}$? Doesn’t seem to improve lattice effectiveness.

Find three balanced integers given half the bits of product?

Do better with another shape?
This allows compression from 2048 bits to 682 bits, with balanced $p, q$.

Minor flaw: uniform random $p_0$ does not produce exactly uniform random integer $p$.
But confirm experimentally that each $p_0$ has good chance of producing at least one $p$.
This implies that each choice of $p$ has probability not far above uniform.

Some open questions:
Find random $p, q \approx 2^{1024}$ given $pq \mod 2^{1500}$? Maybe use higher-dimensional lattices.
Or $p \approx 2^{768}, q \approx 2^{1280}$? Doesn’t seem to improve lattice effectiveness.
Find three balanced integers given half the bits of product?
Do better with another $f$ shape?
Some open questions:

Find random $p, q \approx 2^{1024}$ given $pq \mod 2^{1500}$? Maybe use higher-dimensional lattices.

Or $p \approx 2^{768}, q \approx 2^{1280}$?

Doesn’t seem to improve lattice effectiveness.

Find three balanced integers given half the bits of product?

Do better with another $f$ shape?

Key-generation speed

Start with many $p$’s.

Use trial division etc.

Then try $2^{p-1} \mod 21500$ to find one prime.

Or $2^{768}, 2^{1280}$?

Doesn’t seem to improve lattice effectiveness.

Traditional key generation chooses $p, q$ independently.

Faster, slightly non-uniform:

build visible primes (Maurer).

If $p$ determines $q$: $2^{12}$ exponentiations.

If $p$ determines $q$: $2^{12}$ exponentiations.
Some open questions:

Find random $p, q \approx 2^{1024}$
given $pq \mod 2^{1500}$? Maybe
use higher-dimensional lattices.

Or $p \approx 2^{768}, q \approx 2^{1280}$?
Doesn’t seem to improve
lattice effectiveness.

Find three balanced integers
given half the bits of product?

Do better with another $f$ shape?

Key-generation speed

Start with many $p$’s.
Use trial division etc.
Then try $2^{p-1} \mod p$.
$\approx 2^6$ exponentiations
to find one prime.

Traditional key generation
chooses $p, q$ independently.
$\approx 2^7$ exponentiations.
Faster, slightly non-uniform:
build visible primes (Maurer).

If $p$ determines $q$:
$\approx 2^{12}$ exponentiations.
Some open questions:

- Find random $2^{1024}$ given $\text{mod } 2^{1500}$? Maybe use higher-dimensional lattices.
- Or $2^{768}$, ... effectiveness.
- Find three balanced integers given half the bits of product?
- Do better with another shape?

Key-generation speed

Start with many $p$'s.
Use trial division etc.
Then try $2^{p-1}$ mod $p$.
$\approx 2^6$ exponentiations to find one prime.

Traditional key generation chooses $p, q$ independently.
$\approx 2^7$ exponentiations.
Faster, slightly non-uniform:
build visible primes (Maurer).

If $p$ determines $q$:
$\approx 2^{12}$ exponentiations.

For $f(U) = U$ mod $2^{1008}$:
Each $p$ determines a pool of $2^{16}$ possible $q$'s.
Select randomly from pool until finding a prime:
$\approx 2^7$ exponentiations.

For $f(U) = U$ mod $2^{1350}$:
Obtain pool of pairs $(p, q)$ with all different $p$'s and all different $q$'s.
$\approx 2^{12}$ exponentiations.
Key-generation speed

Start with many $p$'s.
Use trial division etc.
Then try $2^{p-1} \mod p$.
$\approx 2^6$ exponentiations
to find one prime.

Traditional key generation
chooses $p, q$ independently.
$\approx 2^7$ exponentiations.
Faster, slightly non-uniform:
build visible primes (Maurer).

If $p$ determines $q$:
$\approx 2^{12}$ exponentiations.

For $f(U) = U \mod 2^{1008}$:
Each $p$ determines
pool of $2^{16}$ possible $q$'s.
Select randomly from pool
until finding a prime.
$\approx 2^7$ exponentiations.

For $f(U) = U \mod 2^{1350}$:
Obtain pool of pairs $(p, q)$
with all different $p$'s
and all different $q$'s.
$\approx 2^{12}$ exponentiations.
Key-generation speed
Start with many 's.
Use trial division etc.
Then try $2^1 \mod p$.

$2^6$ exponentiations

to find one prime.

Traditional key generation
chooses $p_1$ and $p_2$ independently.

$2^7$ exponentiations.

Faster, slightly non-uniform:
build visible primes (Maurer).
If $d$ determines $p_1$ or $p_2$:
$2^{12}$ exponentiations.

For $f(U) = U \mod 2^{1008}$:
Each $d$ determines a pool of $2^{16}$ possible 's.
Select randomly from pool until finding a prime.
$2^7$ exponentiations.

Or use batch factorization.
Still many exponentiations.
Is there a better approach:
compress slightly more.
Lattice reduction is fast, so can afford many more trials before each exponentiation.

Can use lattice structure to share trial divisions.
Or use batch factorization.

For $f(U) = U \mod 2^{1008}$:
Each $p$ determines
pool of $2^{16}$ possible $q$’s.
Select randomly from pool
until finding a prime.
$\approx 2^7$ exponentiations.

For $f(U) = U \mod 2^{1350}$:
Obtain pool of pairs $(p, q)$
with all different $p$’s
and all different $q$’s.
$\approx 2^{12}$ exponentiations.

Can use lattice structure
to share trial divisions.
Or use batch factorization.
Still many exponentiations.
Is there a better method?

If not, might as well
take opposite approach:
compress slightly more.
Lattice reduction is fast,
so can afford many $p_0$’s
before each exponentiation.
For $2^{1008}$:
- Generates $q_1$'s.
- Select randomly from pool of $2^{16}$ possible $q$'s.
- Until finding a prime.

For $2^{1350}$:
- Generates pool of pairs $(p, q)$ with all different $p$'s and all different $q$'s.
- 212 exponentiations.

Is there a better method?

If not, might as well take opposite approach:
- Compress slightly more.
- Lattice reduction is fast, so can afford many $p_0$'s before each exponentiation.

Protocol violations:
One user generates $U_1$.
Second user sees $j(U_1)$ and generates $U_2$.
Security of $U_2$ was proven assuming uniform random $U_1$.

What if first user cheats and doesn't generate uniform random $U_1$?

Recall half-special NFS: can construct rare values allowing easier factorization.
Can use lattice structure to share trial divisions. Or use batch factorization. Still many exponentiations.

Is there a better method? If not, might as well take opposite approach: compress slightly more. Lattice reduction is fast, so can afford many $p_0$'s before each exponentiation.

Protocol violations
One user generates $U_1$. Second user sees $f(U_1)$ and generates $U_2$. Security of $U_2$ was proven assuming uniform random $U_1$.

What if first user cheats, and doesn't generate uniform random $U_1$? Recall half-special NFS: can construct rare $f$ values allowing easier factorization.
Can use lattice structure to share trial divisions. Or use batch factorization. Still many exponentiations. Is there a better method? If not, might as well compress slightly more.

Lattice reduction is fast, so can afford many 0's before each exponentiation.

Protocol violations
One user generates $U_1$. Second user sees $f(U_1)$ and generates $U_2$.

Security of $U_2$ was proven assuming uniform random $U_1$.

What if first user cheats, and doesn't generate uniform random $U_1$?

Recall half-special NFS: can construct rare $f$ values allowing easier factorization.

One solution is to generate $p_1, q_1, U_1$ publicly from digits of $\pi$: 10th, 20th, 30th, etc. Not random, but conjecturally safe.

Variant: $U_1$ without $p_0$'s exponentiation.

Another solution is to generate $p_1, q_1, U_1$ publicly from SHA-256 output.

Another solution is to generate $p_1, q_1, U_1$ publicly from 1955 RAND tables.
Protocol violations

One user generates $U_1$.
Second user sees $f(U_1)$ and generates $U_2$.

Security of $U_2$ was proven assuming uniform random $U_1$.

What if first user cheats, and doesn’t generate uniform random $U_1$?

Recall half-special NFS: can construct rare $f$ values allowing easier factorization.

One solution is to generate $p_1, q_1, U_1$ publicly from digits of $\pi$:
10th, 20th, 30th, etc.
Not random, but conjecturally safe.
Variant: $U_1$ without $p_1, q_1$.

Another solution is to generate $p_1, q_1, U_1$ publicly from SHA-256 output.

Another solution is to generate $p_1, q_1, U_1$ publicly from 1955 RAND tables.