High-speed cryptographic functions
D. J. Bernstein

Typical Internet protocol:

```
message generator

m

sender

m

network

m'

receiver
```
Typical Internet protocol:

A message generator creates a message $m$, a string of bytes.

Message generator gives $m$ to a sender.

Sender gives $m$ to a network.

Network gives a message $m'$ to a receiver.

Maybe $m' = m$; maybe not.

Maybe network is controlled by an attacker who changed $m$.
A message generator creates a message $m$, a string of bytes.

Message generator gives $m$ to a sender.

Sender gives $m$ to a network.

Network gives a message $m'$ to a receiver.

Maybe $m' = m$; maybe not.
Maybe network is controlled by an attacker who changed $m$ into $m' \neq m$. 
A **message generator** creates a **message** $m$, a string of bytes.

Message generator gives $m$ to a **sender**.

Sender gives $m$ to a **network**.

Network gives a message $m'$ to a **receiver**.

Maybe $m' = m$; maybe not. Maybe network is controlled by an attacker who changed $m$ into $m' \neq m$.  

Protocol eliminating forgeries:

message generator

sender using $r; s$

network

receiver using $r; s$
A message generator creates a message $m$, a string of bytes.

Message generator gives $m$ to a sender.

Sender gives $m$ to a network.

Network gives a message $m'$ to a receiver.

Maybe $m' = m$; maybe not.

Maybe network is controlled by an attacker who changed $m$ into $m' \neq m$.

Protocol eliminating forgeries:

message generator

↓ ↓

sender using $r, s$

↓ ↓

network

↓ ↓

receiver using $r, s$
A message generator creates a message \( m \), a string of bytes. Message generator gives \( m \) to a sender. Sender gives \( m \) to a network. Network gives a message \( m_0 \) to a receiver. Maybe \( m_0 = m \); maybe not. Maybe network is controlled by an attacker who changed \( m \) into \( m' \).

Protocol eliminating forgeries:

1. **Message Generator**
   - Message generator
   - Creates message \( m \)

2. **Sender**
   - Using \( r, s \)
   - Sends \( m \)

3. **Network**
   - \( m, a \)

4. **Receiver**
   - Using \( r, s \)
   - Receives \( m', a' \)

Fix a finite field \( k \).
Typically \( \#k \approx 2^{128} \).

Sender, receiver share a secret:
- Uniform random \((r, s) \in k^2 \times k^\ast \).

Network's function is independent of \((r, s)\).

Sender encodes message \( m \) as polynomial \( m(x) = m' \).
Sender then computes authenticator \( a = m(r) + s \).

Receiver discards \((m', a')\) if \( a' \neq m'(r) + s \).
Fix a finite field $k$. Typically $\#k \approx 2^{128}$.

Sender, receiver share a secret:
uniform random $(r, s) \in k \times k$.

Network’s function $m, a \mapsto m', a'$ is independent of $(r, s)$.

Sender encodes message $m$ as polynomial $\underline{m} \in xk[x]$.
Sender then computes authenticator $a = \underline{m}(r) + s$.

Receiver discards $m', a'$ if $a' \neq \underline{m'}(r) + s$. 

Fix a finite field $k$.
Typically $\#k \approx 2^{128}$.

Sender, receiver share a secret:
uniform random $(r, s) \in k \times k$.
Network's function $m, a \mapsto m', a'$
is independent of $(r, s)$.

Sender encodes message $m$
as polynomial $\overline{m} \in xk[x]$.
Sender then computes
authenticator $a = \overline{m}(r) + s$.

Receiver discards $m', a'$
if $a' \neq \overline{m}'(r) + s$.

If $m' \neq m$ then
$\Pr[\text{receiver accepts } m'] \leq \max\{\deg \overline{m}, \deg \overline{m}'\}$
e.g. $\Pr \leq 2^{-98}$ if $\#k = 2^{128}$ and message degree $\leq 2^{30}$.

Proof: $m' \neq m$ implies $\#k - a' \neq m'$, so $m' - a' \neq m'$.
$\#k$ pairs $(r, s) \in k \times k$ satisfy $a = \overline{m}(r)$.
$\leq \max\{\deg \overline{m}, \deg \overline{m}'\}$ pairs also satisfy $a' = \overline{m}'(r)$.

$\square$
Fix a finite field $k$.
Typically $\#k \approx 2^{128}$.

Sender, receiver share a secret:
uniform random $(r, s) \in k \times k$.
Network’s function $m, a \mapsto m', a'$
is independent of $(r, s)$.

Sender encodes message $m$
as polynomial $m \in xk[x]$.
Sender then computes
authenticator $a = m(r) + s$.
Receiver discards $m', a'$
if $a' \neq m'(r) + s$.

If $m' \neq m$ then
$\Pr[\text{receiver accepts } m']$
$\leq \max \{\deg m, \deg m'\} / \#k$.

e.g. $\Pr \leq 2^{-98}$ if $\#k = 2^{128}$
and message degree $\leq 2^{30}$.

Proof: $m' \neq m$ in $xk[x]$ so $m' - a' \neq m - a$ in $k[x]$.
$\#k$ pairs $(r, s) \in k \times k$
satisfy $a = m(r) + s$.
$\leq \max \{\deg m, \deg m'\}$ pairs also satisfy $a' = m'(r) + s$.  
\[\square\]
Fix a finite field $k$.
Typically $|k| = 2^{128}$.
Sender, receiver share a secret: $(r, s) \in k \times k$.
Compute $m, a \mapsto m', a'$ of $(r, s)$.
If $m' \neq m$ then
$\Pr[\text{receiver accepts } m'] \leq \max\{\deg m, \deg m'\}/|k|$.
e.g. $\Pr \leq 2^{-98}$ if $|k| = 2^{128}$ and message degree $\leq 2^{30}$.

Proof: $m' \neq m$ in $k[x]$ so $m' - a' \neq m - a$ in $k[x]$.
$|k|$ pairs $(r, s) \in k \times k$ satisfy $a = m(r) + s$.
$\leq \max\{\deg m, \deg m'\}$ pairs also satisfy $a' = m'(r) + s$. 

Many messages, unprotected:
message generator
$m_1; m_2; \ldots$
sender
$1, m_1; 2, m_2; \ldots$
network
$n_1', m_1'; n_2', m_2'; \ldots$
receiver
If $m' \neq m$ then

\[ \Pr[\text{receiver accepts } m'] \leq \max\{\deg m, \deg m'\} / \#k. \]

e.g. $\Pr \leq 2^{-98}$ if $\#k = 2^{128}$ and message degree $\leq 2^{30}$.

Proof: $m' \neq m$ in $xk[x]$ so $m' - a' \neq m - a$ in $k[x]$.

$\#k$ pairs $(r, s) \in k \times k$ satisfy $a = m(r) + s$.

$\leq \max\{\deg m, \deg m'\}$ pairs also satisfy $a' = m'(r) + s$.

Many messages, unprotected:

1. message generator
2. $m_1; m_2; \ldots$
3. sender
4. $1, m_1; 2, m_2; \ldots$
5. network
6. $n'_1, m'_1; n'_2, m'_2; \ldots$
7. receiver
If $m_0 \neq m$ then
Pr[receiver accepts $m_0$] » max $\bar{\deg} m, \bar{\deg} m_0 = \# k$.

E.g. Pr > $2^{98}$ if $\# k = 2^{128}$ and message degree > $2^{30}$.

Proof:
$m_0 = m$ in $xk[x]$
$- a$ in $k[x]$.
$k \times k$
$+ s$.

$\deg m'$ pairs
$m'(r) + s$.

Many messages, unprotected:

message generator
\[
\begin{array}{c}
\downarrow \\
m_1; m_2; \ldots
\end{array}
\]
sender
\[
\begin{array}{c}
\downarrow \\
1, m_1; 2, m_2; \ldots
\end{array}
\]
network
\[
\begin{array}{c}
\downarrow \\
n'_1, m'_1; n'_2, m'_2; \ldots
\end{array}
\]
receiver

Many messages, protected:

message generator
\[
\begin{array}{c}
\downarrow \\
m_1; m_2; \ldots
\end{array}
\]
sender using $r; s$
\[
\begin{array}{c}
\downarrow \\
1, m_1; a_1; 2, m_2; a_2; \ldots
\end{array}
\]
network
\[
\begin{array}{c}
\downarrow \\
n'_1, m'_1; n'_2, m'_2; \ldots
\end{array}
\]
receiver using $r; s$
Many messages, unprotected:

```
message generator

```

```
m_1; m_2; ...
```

```
sender
```

```
1, m_1; 2, m_2; ...
```

```
network
```

```
n'_1, m'_1; n'_2, m'_2; ...
```

```
receiver
```

Many messages, protected:

```
message generator

```

```
m_1; m_2; ...
```

```
sender using r, s
```

```
1, m_1, a_1; 2, m_2, a_2; ...
```

```
network
```

```
n'_1, m'_1, a'_1; n'_2, m'_2, a'_2; ...
```

```
receiver using r, s
```
Many messages, unprotected:

message generator

\( m_1; m_2; \ldots \)

\( \downarrow \)

sender

\( 1; m_1; 2; m_2; \ldots \)

\( \downarrow \)

network

\( n_0; m_0; 1; n_0; 2; m_0; 2; \ldots \)

\( \downarrow \)

receiver

Many messages, protected:

message generator

\( m_1; m_2; \ldots \)

\( \downarrow \)

sender using \( r, s \)

\( 1, m_1, a_1; 2, m_2, a_2; \ldots \)

\( \downarrow \)

network

\( n'_1, m'_1, a'_1; n'_2, m'_2, a'_2; \ldots \)

\( \downarrow \)

receiver using \( r, s \)

Secret here is uniform random \( (r, s) \in k \times k^{\{1,2\}} \)

i.e., \( r \in k; s(1) \in k \)

e.g. 128000 secret bits to handle 999 messages if \( \# k = 2^{128} \).

Sender transmits \( n, m, \overline{m(r)} + s(n) \).

Receiver discards if \( a' \neq \overline{m'(r)} + s(n') \).

Forged \( n', m', a' \) has negligible chance of being accepted.
Many messages, protected:

- **Message generator**
  - $m_1; m_2; \ldots$
- **Sender using $r, s$**
  - $1, m_1, a_1; 2, m_2, a_2; \ldots$
- **Network**
  - $n'_1, m'_1, a'_1; n'_2, m'_2, a'_2; \ldots$
- **Receiver using $r, s$**

Secret here is uniform random
$(r, s) \in k \times k^{\{1, 2, \ldots\}}$;
i.e., $r \in k; s(1) \in k; s(2) \in k; \ldots$
e.g. 128000 secret bits
to handle 999 messages
if $\#k = 2^{128}$.

Sender transmits $n$th message $m$ as $n, m, m(r) + s(n)$.
Receiver discards $n', m', a'$
if $a' \neq m'(r) + s(n')$.
Forged $n', m', a'$ has negligible chance of being accepted.
Many messages, protected:
m_1; m_2; ::; sender using r_; s_1; m_1; a_1; 2; m_2; a_2; ::; network
n_0; m_0; a_0; n_0; m_0; a_0; ::;
receiver using r_; s

Secret here is uniform random
(r, s) ∈ k × k^_;
i.e., r ∈ k; s(1) ∈ k; s(2) ∈ k; . . .
e.g. 128000 secret bits
to handle 999 messages
if #k = 2^128.

Sender transmits n_th message m
as n, m, m(r) + s(n).

Receiver discards n', m', a'
if a' ≠ m(r) + s(n').

Forged n', m', a' has negligible
chance of being accepted.

How did sender, receiver
create and share r_; s?
Must have had previous channel
providing secrecy, authenticity.

Why not use that channel
for new messages?

Answer 1: Extend security
through time. Previous channel
can disappear after sending r_; s.
New channel sends new messages.

Answer 2: Expand bandwidth.
Messages can be much longer
than r_; s.

How did sender, receiver
create and share r_; s?
Must have had previous channel
providing secrecy, authenticity.

Why not use that channel
for new messages?

Answer 1: Extend security
through time. Previous channel
can disappear after sending r_; s.
New channel sends new messages.

Answer 2: Expand bandwidth.
Messages can be much longer
than r_; s.
Secret here is uniform random 
\((r, s) \in k \times k^{\{1,2,\ldots\}}\);
i.e., \(r \in k; s(1) \in k; s(2) \in k; \ldots\).
e.g. 128000 secret bits
to handle 999 messages
if \(#k = 2^{128}\).

Sender transmits \(n\)th message \(m\)
as \(n, m, \underline{m}(r) + s(n)\).
Receiver discards \(n', m', a'\)
if \(a' \neq \underline{m}'(r) + s(n')\).
Forged \(n', m', a'\) has negligible chance of being accepted.

How did sender, receiver create and share \(r, s\)?
Must have had previous channel providing secrecy, authenticity.

Why not use that channel for new messages?

Answer 1: Extend security through time. Previous channel can disappear after sending \(r, s\).
New channel sends new messages.

Answer 2: Expand bandwidth.
Messages can be much longer than \(r, s\).
For uniform random $(r, s) \in k \times k \{1, 2, \ldots\}$;
\[ r^2 \in k; s(2) \in k; \ldots \]
determine bits
for messages

For $n$th message $m$ as
\[ m = r \oplus s(n). \]

For $n'$, $m'$, $a'$
\[ m' = a' \oplus s(n'). \]

Has negligible
chance of being accepted.

How did sender, receiver
create and share $r$, $s$?
Must have had previous channel providing secrecy, authenticity.

Why not use that channel
for new messages?

Answer 1: Extend security
through time. Previous channel
can disappear after sending $r$, $s$.
New channel sends new messages.

Answer 2: Expand bandwidth.
Messages can be much longer
than $r$, $s$.

For $b$-bit security, $\frac{\log \# k e}{b} = c$ messages, total length $d$:

transmit $r, s$ through
old channel providing
secrecy and authenticity
for $b + bc$ bits
↓
transmit $m_1, m_2, \ldots$ through
new channel providing
authenticity
for $d$ bits

For $b$-bit security, $\frac{\log \# k e}{b} = c$ messages, total length $d$:
transmit $r, s$ through
old channel providing
secrecy and authenticity
for $b + bc$ bits
↓
transmit $m_1, m_2, \ldots$ through
new channel providing
authenticity
for $d$ bits

How did sender, receiver
create and share $r$, $s$?
Must have had previous channel providing secrecy, authenticity.

Why not use that channel
for new messages?

Answer 1: Extend security
through time. Previous channel
can disappear after sending $r$, $s$.
New channel sends new messages.

Answer 2: Expand bandwidth.
Messages can be much longer
than $r$, $s$. 

transmit $m_1, m_2, \ldots$
new channel providing
authenticity
for $d$ bits
How did sender, receiver create and share \( r, s \)?
Must have had previous channel providing secrecy, authenticity.

Why not use that channel for new messages?

**Answer 1:** Extend security through time. Previous channel can disappear after sending \( r, s \). New channel sends new messages.

**Answer 2:** Expand bandwidth. Messages can be much longer than \( r, s \).

For \( b \)-bit security, \( \lceil \log \#k \rceil = b \), \( c \) messages, total length \( d \):

- Transmit \( r, s \) through old channel providing secrecy and authenticity for \( b + bc \) bits
- Transmit \( m_1; m_2; \ldots \) through new channel providing authenticity for \( d \) bits
For $b$-bit security, $\lceil \log \#k \rceil = b$, $c$ messages, total length $d$:

transmit $r$, $s$ through old channel providing secrecy and authenticity for $b + bc$ bits

transmit $m_1; m_2; \ldots$ through new channel providing authenticity for $d$ bits

Authenticated-encryption variant using $n, ((m, m(r)) + s(n))$:

transmit $r$, $s$ through old channel providing secrecy and authenticity for $b + bc$ bits

transmit $m_1; m_2; \ldots$ through new channel providing secrecy and authenticity for $d$ bits
For \( b \)-bit security, \([\lg \#k] = b\),  
c messages, total length \( d \):

- Transmit \( r, s \) through old channel providing secrecy and authenticity for \( b + bc \) bits

\[
\text{transmit } r, s \text{ through old channel providing secrecy and authenticity for } b + bc \text{ bits}
\]

- Transmit \( m_1; m_2; \ldots \) through new channel providing authenticity for \( d \) bits

\[
\text{transmit } m_1; m_2; \ldots \text{ through new channel providing authenticity for } d \text{ bits}
\]

Authenticated-encryption variant using \( n, ((m, m(r)) + s(n)) \):

- Transmit \( r, s \) through old channel providing secrecy and authenticity for \( b + bc + d \) bits

\[
\text{transmit } r, s \text{ through old channel providing secrecy and authenticity for } b + bc + d \text{ bits}
\]

- Transmit \( m_1; m_2; \ldots \) through new channel providing secrecy and authenticity for \( d \) bits

\[
\text{transmit } m_1; m_2; \ldots \text{ through new channel providing secrecy and authenticity for } d \text{ bits}
\]
For $b$-bit security, $d \lg \#k = b$, total length $d$:

transmit $r, s$ through old channel providing secrecy and authenticity for $b + bc$ bits

transmit $m_1, m_2, \ldots$ through new channel providing authenticity for $d$ bits

Authenticated-encryption variant using $n, ((m, \overline{m}(r)) + s(n))$:

transmit $r, s$ through old channel providing secrecy and authenticity for $b + bc + d$ bits

transmit $m_1, m_2, \ldots$ through new channel providing secrecy and authenticity for $d$ bits

Can multiply in $k$ using $b^{1+o(1)}$ bit operations; more precisely, $b(\lg b)^{1+o(1)}$.

Can evaluate $\overline{m}(r)$ using $b(\lg b)^{1+o(1)}$ bit operations for each $b$-bit block.

Overall $(bc + d)(\lg b)^{1+o(1)}$ bit operations.

Normally $d$ dominates $bc$, so $(\lg b)^{1+o(1)}$ bit operations for each message bit.
Authenticated-encryption variant using $n, ((m, \overline{m}(r)) + s(n))$:

- transmit $r, s$ through old channel providing secrecy and authenticity for $b + bc + d$ bits

transmit $m_1; m_2; \ldots$ through new channel providing secrecy and authenticity for $d$ bits

Can multiply in $k$ using $b^{1+o(1)}$ bit operations; more precisely, $b(\lg b)^{1+o(1)}$.

Can evaluate $\overline{m}(r)$ using $b(\lg b)^{1+o(1)}$ bit operations for each $b$-bit block of $m$.

Overall $(bc + d)(\lg b)^{1+o(1)}$ bit operations.

Normally $d$ dominates $bc$, so $(\lg b)^{1+o(1)}$ bit operations for each message bit.
Authenticated-encryption variant using \( n; (m;m (r)) + s(n)) \):
transmit \( r;s \) through old channel providing secrecy and authenticity for \( b + bc + d \) bits:
\[
\begin{align*}
\text{transmit } m_1; m_2; \ldots \text{ through new channel providing secrecy and authenticity for } d \text{ bits.}
\end{align*}
\]

Can multiply in \( k \) using \( b \) \( 1+o(1) \) bit operations; more precisely, \( b \lg b \) \( 1+o(1) \).
Can evaluate \( m(r) \) using \( b \lg b \) \( 1+o(1) \) bit operations.
Overall \( (bc + d)(\lg b) \) \( 1+o(1) \) bit operations.
Normally \( d \) dominates \( bc \), so \( \lg b \) \( 1+o(1) \) bit operations for each message bit.

Survey of alternatives: Sections 8–10 of papers.html#hash127.
See cr.yp.to/mac.html and papers.html#poly1305.

128-bit coefficients of \( m \);
\( k = \mathbb{Z}/(2^{130} - 5) \);
8–10 CPU cycles per bit.

Can analyze cost more precisely.
Can modify \( m(r) \) to improve constants in cost.
Can account for real CPUs.
Can focus on useful \( b \).
Can modify \( m(r) \) to improve cost.

Can focus on useful \( b \).
Can modify \( m(r) \) to improve cost.
Can account for real CPUs.
Can analyze cost more precisely.
Can modify \( m(r) \) to improve constants in cost.
Can multiply in $k$ using $b^{1+o(1)}$ bit operations; more precisely, $b(\lg b)^{1+o(1)}$.

Can evaluate $m(r)$ using $b(\lg b)^{1+o(1)}$ bit operations for each $b$-bit block of $m$.

Overall $(bc + d)(\lg b)^{1+o(1)}$ bit operations.

Normally $d$ dominates $bc$, so $(\lg b)^{1+o(1)}$ bit operations for each message bit.

Can analyze cost more precisely. Can modify $m(r)$ to improve constants in cost. Can account for real CPUs. Can focus on useful $b$.

Speed records: Poly1305. See cr.yp.to/mac.html and papers.html#poly1305. 128-bit coefficients of $m$; $k = \mathbb{Z}/(2^{130} - 5)$; restricted $r$; $\approx 0.5$ CPU cycles per bit.

Survey of alternatives: Sections 8–10 of papers.html#hash127.
Can multiply in $k$ using $b^{1+o(1)}$ bit operations; more precisely, $b^{(\lg b)^{1+o(1)}}$.

Can evaluate $m(r)$ using $b^{(\lg b)^{1+o(1)}}$ bit operations for each block of $m$.

Can analyze cost more precisely. Can modify $m(r)$ to improve constants in cost. Can account for real CPUs. Can focus on useful $b$.

Speed records: Poly1305. See cr.yp.to/mac.html and papers.html#poly1305.

128-bit coefficients of $m$; $k = \mathbb{Z}/(2^{130} - 5)$; restricted $r$; $\approx 0.5$ CPU cycles per bit.

Survey of alternatives: Sections 8–10 of papers.html#hash127.

We’ll see better protocols that dramatically reduce requirements on old channel:

1. Reduce bandwidth far below $b + bc + d$ bits.

2. Eliminate secrecy.

Disadvantage: no known way to prove security of better protocols.
Can analyze cost more precisely. Can modify $m(r)$ to improve constants in cost. Can account for real CPUs. Can focus on useful $b$.

Speed records: Poly1305. See cr.yp.to/mac.html and papers.html#poly1305. 128-bit coefficients of $m$; $k = \mathbb{Z}/(2^{130} - 5)$; restricted $r$; $\approx 0.5$ CPU cycles per bit.

Survey of alternatives: Sections 8–10 of papers.html#hash127.

We’ll see better protocols that dramatically reduce requirements on old channel:

1. Reduce bandwidth far below $b + bc + d$ bits.
2. Eliminate secrecy.

Disadvantage: no known way to prove security of better protocols.
We’ll see better protocols that dramatically reduce requirements on old channel:

1. Reduce bandwidth far below $b + bc + d$ bits.
2. Eliminate secrecy.

Disadvantage: no known way to prove security of better protocols.

How to reduce bandwidth?
Expand a short shared secret into a long shared secret.

\[ 4^t \mod q, \]
\[ 4^{tu_0} \mod q, \]
\[ 4^{tu_1} \mod q, \]
\[ 4^{tu_0u_1} \mod q, \]
\[ 4^{tu_2} \mod q, \]
\[ 4^{tu_0u_2} \mod q, \]
\[ 4^{tu_1u_2} \mod q, \]
\[ 4^{tu_0u_1u_2} \mod q, \]

etc., where $q = 2^{2000} - 1553657$. 

We’ll see better protocols that dramatically reduce requirements in cost.

Can modify $m(r)$ to improve constants in cost.

Can account for real CPUs.

Can focus on useful $b$.

Poly1305.

See cr.yp.to/mac.html and papers.html#poly1305.

128-bit coefficients of $m$; restricted $r$; ı0: 5 CPU cycles per bit.

Speed records: Poly1305.

Survey of alternatives: Sections 8–10 of papers.html#hash127.

Expand a short shared secret into a long shared secret.

\[ 4^t \mod q, \]
\[ 4^{tu_0} \mod q, \]
\[ 4^{tu_1} \mod q, \]
\[ 4^{tu_0u_1} \mod q, \]
\[ 4^{tu_2} \mod q, \]
\[ 4^{tu_0u_2} \mod q, \]
\[ 4^{tu_1u_2} \mod q, \]
\[ 4^{tu_0u_1u_2} \mod q, \]

etc., where $q = 2^{2000} - 1553657$. 

We’ll see better protocols that dramatically reduce requirements on old channel:

1. Reduce bandwidth far below $b + bc + d$ bits.
2. Eliminate secrecy.

Disadvantage: no known way to prove security of better protocols.
We’ll see better protocols that dramatically reduce requirements on old channel:

1. Reduce bandwidth far below $b + bc + d$ bits.
2. Eliminate secrecy.

Disadvantage: no known way to prove security of better protocols.

How to reduce bandwidth?

Expand a short shared secret into a long shared secret.

e.g. Expand $t, u_0, u_1, \ldots$ into $4^t \mod q,$ $4^{tu_0} \mod q,$ $4^{tu_1} \mod q,$ $4^{tu_0u_1} \mod q,$ $4^{tu_2} \mod q,$ $4^{tu_0u_2} \mod q,$ $4^{tu_1u_2} \mod q,$ $4^{tu_0u_1u_2} \mod q,$ etc., where $q = 2^{2000} - 1553657.$
We'll see better protocols that dramatically reduce requirements on old channel:

1. Reduce bandwidth far below $b + b + d$ bits.
2. Eliminate secrecy.

Disadvantage: no known way to prove security of better protocols.

How to reduce bandwidth?

Expand a short shared secret into a long shared secret.

e.g. Expand $t, u_0, u_1, \ldots$ into $\begin{align*}
4^t & \pmod{q}, \\
4^{tu_0} & \pmod{q}, \\
4^{tu_1} & \pmod{q}, \\
4^{tu_0u_1} & \pmod{q}, \\
4^{tu_2} & \pmod{q}, \\
4^{tu_0u_2} & \pmod{q}, \\
4^{tu_1u_2} & \pmod{q}, \\
4^{tu_0u_1u_2} & \pmod{q}, \\
\end{align*}$

etc., where $q = 2^{2000} - 1553657$.

Conjecture: Hard to distinguish this expanded secret from a uniform random sequence of squares modulo $q$.

Could try to do it by computing discrete logs modulo $q$:

extract $t$ from $4^t \pmod{q}$,

extract $tu_0$ from $4^{tu_0} \pmod{q}$, and see $(t)(tu_0u_1u_2) = (tu_0u_1u_2)$.

But discrete logs seem hard!

Thus (e.g.) bottom halves seem hard to distinguish from a uniform random string.
How to reduce bandwidth?

Expand a short shared secret into a long shared secret.

E.g. Expand $t, u_0, u_1, \ldots$ into

$$4^t \mod q,$$
$$4^{tu_0} \mod q,$$
$$4^{tu_1} \mod q,$$
$$4^{tu_0u_1} \mod q,$$
$$4^{tu_2} \mod q,$$
$$4^{tu_0u_2} \mod q,$$
$$4^{tu_1u_2} \mod q,$$
$$4^{tu_0u_1u_2} \mod q,$$

etc., where $q = 2^{2000} - 1553657$.

Conjecture: Hard to distinguish this expanded secret from a uniform random sequence of squares modulo $q$.

Could try to do it by computing discrete logs modulo $q$:

extract $t$ from $4^t \mod q$,

$tu_0$ from $4^{tu_0} \mod q$, etc.,

and see $(t)(tu_0u_1) = (tu_0)(tu_1)$.

But discrete logs seem hard!

Thus (e.g.) bottom halves seem hard to distinguish from a uniform random string.
Conjecture: Hard to distinguish this expanded secret from a uniform random sequence of squares modulo $q$.

Could try to do it by computing discrete logs modulo $q$:
extract $t$ from $4^t \mod q$, $tu_0$ from $4^{tu_0} \mod q$, etc., and see $(t)(tu_0u_1) = (tu_0)(tu_1)$.
But discrete logs seem hard!

Thus (e.g.) bottom halves seem hard to distinguish from a uniform random string.

For $b$ bits of security, $b \ll 1$:
Fix $q$ with $q, (q - 1) = 2$ prime and with $\log_2 q \in b$; more precisely, with
$6.8 \ldots \log_2 q \log_2 \log_2 q$.

Transmit short shared secret:
independent uniform random $t, u_0, u_1, \ldots \in \{1, \ldots, 2^{2000} - 1553657\}$.

These sizes just barely resist fastest discrete-log methods that we know.
Conjecture: Hard to distinguish this expanded secret from a uniform random sequence of squares modulo $q$.

Could try to do it by computing discrete logs modulo $q$: extract $t$ from $4^t \mod q$, $tu_0$ from $4^{tu_0} \mod q$, etc., and see $(t)(tu_0u_1) = (tu_0)(tu_1)$. But discrete logs seem hard!

Thus (e.g.) bottom halves seem hard to distinguish from a uniform random string.

For $b$ bits of security, $b \to \infty$:

Fix $q$ with $q, (q - 1)/2$ prime and with $\lg q \in b^{3+o(1)}$; more precisely, with $6.8 \ldots (\lg q)(\lg \log q)^2 \approx b^3$.

Transmit short shared secret: independent uniform random $t, u_0, u_1, \ldots \in \{1, 2, \ldots, 2^{2b}\}$.

These sizes just barely resist fastest discrete-log methods that we know.
Conjecture: Hard to distinguish this expanded secret from a uniform random sequence of squares modulo \(q\).

Could try to do it by computing discrete logs modulo \(q\):

\[
\begin{align*}
&\text{extract } t \text{ from } 4^t \mod q, \\
&\text{tu} \text{ from } 4^{tu} \mod q, \text{ etc.,}
\end{align*}
\]

and see

\[
(t)(tu_0u_1) = (tu_0)(tu_1).
\]

But discrete logs seem hard!

Thus (e.g.) bottom halves seem hard to distinguish from a uniform random string.

For \(b\) bits of security, \(b \rightarrow \infty\):

Fix \(q\) with \((q - 1)/2\) prime and with \(\log q \in b^{3+o(1)}\);

more precisely, with

\[
6.8 \ldots (\log q)(\log \log q)^2 \approx b^3.
\]

Transmit short shared secret:

independent uniform random \(t, u_0, u_1, \ldots \in \{1, 2, \ldots, 2^{2b}\}\).

These sizes just barely resist fastest discrete-log methods that we know.

Expand \(t, u_0, u_1, \ldots\) into \(4^t \mod q, \ldots, 4^{tu}\), e.g. Expand \(t, u_0, u_1, \ldots, u_{63}\) into \(2^{64}\) integers modulo \(q\).

Extract bottom \(\lfloor \log_2 \lfloor \log_2 \rfloor \rfloor \) bits of each integer.

Compute results sequentially:

Only \(O(b)\) mults \(\mod q\) for each integer, so

\[
b(\log b)^{1+o(1)} \text{ bit ops per bit.}
\]

Random access is slow: \(\approx b^{4+o(1)}\) bit ops.
For $b$ bits of security, $b \to \infty$:

Fix $q$ with $q, (q - 1)/2$ prime and with $\lg q \in b^3 + o(1)$; more precisely, with

$6.8 \ldots (\lg q)(\lg \lg q)^2 \approx b^3$.

Transmit short shared secret: independent uniform random $t, u_0, u_1, \ldots \in \{1, 2, \ldots, 2^{2b}\}$.

These sizes just barely resist fastest discrete-log methods that we know.

Expand $t, u_0, u_1, \ldots$ into $4^t \mod q, \ldots, 4^{tu_0u_2} \mod q, \ldots$ e.g. Expand $t, u_0, u_1, \ldots, u_{63}$ into $2^{64}$ integers modulo $q$.

Extract bottom $\lceil (1/2) \lg q \rceil$ bits of each integer.

Compute results sequentially. Only $O(b)$ mults mod $q$ for each integer, so $b(\lg b)^{1+o(1)}$ bit ops per bit.

Random access is slow: $\geq b^{4+o(1)}$ bit ops.
For $b$ bits of security, $b \rightarrow \infty$:

- Fix $q$ with $q \approx (q - 1)/2$ prime
- with $\text{lg } q \approx 3 + o(1)$.

more precisely, with

$6 \ldots 8 \ldots$ $\approx (\text{lg } q) (\text{lg } \log q)^2 \approx b^3$.

Transmit short shared secret:

- independent uniform random $t; u_0; u_1; \ldots$ $\in \{1, 2, \ldots, 2^{2b}\}$.

These sizes just barely resist fastest discrete-log methods that we know.

Expand $t; u_0; u_1; \ldots$ into $4^t \mod q; \ldots, 4^{tu_0u_2} \mod q, \ldots$.

e.g. Expand $t; u_0; u_1; \ldots, u_{63}$ into $2^{64}$ integers modulo $q$.

Extract bottom \(\lfloor (1/2) \text{lg } q \rfloor\) bits of each integer.

Compute results sequentially.

Only $O(b)$ mults mod $q$ for each integer, so $b(\text{lg } b)^{1 + o(1)}$ bit ops per bit.

Random access is slow:

$\geq b^{4 + o(1)}$ bit ops.

Do better by replacing $(\mathbb{Z}/q)^*$ with $E(\mathbb{Z}/q)$ for a safe elliptic curve $E$.

Discrete logs in $E(\mathbb{Z}/q)$ seem relatively difficult, so can take $q$ smaller:

- specifically, $\text{lg } q \approx 2^b$.

Much faster random access:

$b^{2 + o(1)}$ bit ops.

Sequential access again takes $b(\text{lg } b)^{1 + o(1)}$ bit ops.
Expand $t, u_0, u_1, \ldots$ into
$4^t \mod q, \ldots, 4^{tu_0u_2} \mod q, \ldots$
e.g. Expand $t, u_0, u_1, \ldots, u_{63}$
into $2^{64}$ integers modulo $q$.

Extract bottom $\lceil (1/2) \lg q \rceil$ bits
of each integer.

Compute results sequentially.
Only $O(b)$ mults mod $q$
for each integer, so
$b(\lg b)^{1+o(1)}$ bit ops per bit.

Random access is slow:
$\geq b^{4+o(1)}$ bit ops.

Do better by replacing $(\mathbb{Z}/q)^*$ with $E(\mathbb{Z}/q)$ for a safe elliptic curve $E$.

Discrete logs in $E(\mathbb{Z}/q)$
seem relatively difficult,
so can take $q$ smaller:
specifically, $\lg q \approx 2b$.

Much faster random access:
$b^{2+o(1)}$ bit ops.

Sequential access again takes
$b(\lg b)^{1+o(1)}$ bit ops per bit.
Expand $t; u_0; u_1; \ldots; u_{63}$ into $4$ integers modulo $q$. 

Do better by replacing $(\mathbb{Z}/q)^*$ with $E(\mathbb{Z}/q)$ for a safe elliptic curve $E$. Discrete logs in $E(\mathbb{Z}/q)$ seem relatively difficult, so can take $q$ smaller: specifically, $\lg q \approx 2b$. 

Much faster random access: $b^{2+o(1)}$ bit ops. Sequential access again takes $b(\lg b)^{1+o(1)}$ bit ops per bit. 

Many choices of expansion functions ("stream ciphers"). Fastest expansion functions don't have discrete-log structure. Speed records: see eSTREAM, www.ecrypt.eu.org/stream. Often $< 1$ CPU cycle per bit; random access $< 1000$ cycles.
Do better by replacing \((\mathbb{Z}/q)^*\) with \(E(\mathbb{Z}/q)\) for a safe elliptic curve \(E\).

Discrete logs in \(E(\mathbb{Z}/q)\) seem relatively difficult, so can take \(q\) smaller: specifically, \(\lg q \approx 2b\).

Much faster random access: \(b^{2+o(1)}\) bit ops.

Sequential access again takes \(b(\lg b)^{1+o(1)}\) bit ops per bit.

Can do much better. Maybe non-constant speedups; certainly constant speedups; just one \(b\)-bit secret instead of several \(2b\)-bit secrets \(t, u_0, u_1, \ldots\); focus on useful \(b\); etc.

Many choices of expansion functions ("stream ciphers"). Fastest expansion functions don’t have discrete-log structure.

Speed records: see eSTREAM, www.ecrypt.eu.org/stream. Often < 1 CPU cycle per bit; random access < 1000 cycles.
Can do much better. Maybe non-constant speedups; certainly constant speedups; just one \( b \)-bit secret instead of several \( 2b \)-bit secrets \( t, u_0, u_1, \ldots \); focus on useful \( b \); etc.

Many choices of expansion functions ("stream ciphers"). Fastest expansion functions *don’t* have discrete-log structure.

Speed records: see eSTREAM, www.ecrypt.eu.org/stream. Often < 1 CPU cycle per bit; random access < 1000 cycles.

Channels at this point:
transmit secret through old channel providing secrecy and authenticity for \( b \) bits
\[ \downarrow \downarrow \]
transmit \( m_1; m_2; \ldots \) through new channel providing secrecy and authenticity for \( d \) bits

Channels at this point:
transmit secret through old channel providing secrecy and authenticity for \( b \) bits
\[ \downarrow \downarrow \]
transmit \( m_1; m_2; \ldots \) through new channel providing secrecy and authenticity for \( d \) bits
Can do much better. Maybe non-constant speedups; certainly constant speedups; just one $b$-bit secret instead of several $2b$-bit secrets $t, u_0, u_1, \ldots$; focus on useful $b$; etc.

Many choices of expansion functions ("stream ciphers"). Fastest expansion functions *don't* have discrete-log structure.

Speed records: see eSTREAM, www.ecrypt.eu.org/stream. Often $< 1$ CPU cycle per bit; random access $< 1000$ cycles.

Channels at this point:

- transmit secret through old channel providing secrecy and authenticity for $b$ bits
- transmit $m_1; m_2; \ldots$ through new channel providing secrecy and authenticity for $d$ bits
Channels at this point:

transmit secret through old channel providing secrecy and authenticity for \( b \) bits

transmit \( m_1; m_2; \ldots \) through new channel providing secrecy and authenticity for \( d \) bits

How to use old channel providing only authenticity, not secrecy?

Sender generates secret \( \sigma \), sends public key \( \tau \), \( \mod q \) through old channel.

Receiver generates secret \( \tau \), sends public key \( \sigma \), \( \mod q \) back through old channel.

Sender and receiver compute \( 4^{\sigma \tau} \mod q \), extract \( b \) bits, expand into long shared secret.
Channels at this point:

transmit secret through old channel providing secrecy and authenticity for $b$ bits

↓

transmit $m_1; m_2; \ldots$ through new channel providing secrecy and authenticity for $d$ bits

How to use old channel providing only authenticity, not secrecy?

Sender generates secret $\sigma$, sends \textbf{public key} $4^\sigma \mod q$ through old channel.

Receiver generates secret $\tau$, sends public key $4^\tau \mod q$ back through old channel.

Sender and receiver now compute $4^{\sigma\tau} \mod q$, extract $b$ bits, expand into long shared secret.
How to use old channel providing only authenticity, not secrecy?

Sender generates secret $\sigma$, sends **public key** $4^\sigma \mod q$ through old channel.

Receiver generates secret $\tau$, sends public key $4^\tau \mod q$ back through old channel.

Sender and receiver now compute $4^{\sigma \tau} \mod q$, extract $b$ bits, expand into long shared secret.

Conjecture: Given $4^\tau \mod q$, hard to distinguish $4^{\sigma \tau} \mod q$ from uniform random?

As before, can distinguish by computing discrete logs, but that seems hard.

As before, reduce costs by switching to elliptic curves.

Many choices of functions?

No! Need discrete-log structure.
How to use old channel providing only authenticity, not secrecy?

Sender generates secret $\sigma$, sends public key $4^\sigma \mod q$ through old channel.

Receiver generates secret $\tau$, sends public key $4^\tau \mod q$ back through old channel.

Sender and receiver now compute $4^{\sigma\tau} \mod q$, extract $b$ bits, expand into long shared secret.

Conjecture: Given $4^\sigma \mod q$, $4^\tau \mod q$, hard to distinguish $4^{\sigma\tau} \mod q$ from uniform random square mod $q$.

As before, can distinguish by computing discrete logs, but that seems hard.

As before, reduce costs by switching to elliptic curves.

Many choices of functions?
No! Need discrete-log structure.
How to use old channel providing only authenticity, not secrecy?

Sender generates secret $\sigma$, sends public key $4^\sigma \mod q$ through old channel.

Receiver generates secret $\tau$, sends public key $4^\tau \mod q$ back through old channel.

Sender and receiver now compute $4^{\sigma\tau} \mod q$, extract $b$ bits, expand into long shared secret.

Conjecture: Given $4^\sigma \mod q$, $4^\tau \mod q$, hard to distinguish $4^{\sigma\tau} \mod q$ from uniform random square $\mod q$.

As before, can distinguish by computing discrete logs, but that seems hard.

As before, reduce costs by switching to elliptic curves.

Many choices of functions?
No! Need discrete-log structure.

Again improve cost constants, focus on useful $b$, etc.

Speed records: Curve25519.
See cr.yp.to/ecdh.html and papers.html#curve25519.

Curve $y^2 = x^3 + 486662x + 1$ \mod $2^{255} - 19$; each key < 1000000 CPU cycles.

New records from Jacobians of genus-2 hyperelliptic curves?
Come to ECC 2006 in Toronto.

www.fields.utoronto.ca/programs/scientific/06-07/crypto/
Conjecture: Given $4^\sigma \mod q$, $4^\tau \mod q$, hard to distinguish $4^{\sigma \tau} \mod q$ from uniform random square mod $q$.

As before, can distinguish by computing discrete logs, but that seems hard.

As before, reduce costs by switching to elliptic curves.

Many choices of functions?
No! Need discrete-log structure.

Again improve cost constants, focus on useful $b$, etc.

Speed records: Curve25519. See cr.yp.to/ecdh.html and papers.html#curve25519. Curve $y^2 = x^3 + 486662x^2 + x \mod 2^{255} - 19$; eliminate $y$; $< 1000000$ CPU cycles.

Again improve cost constants, focus on useful $b$, etc.

Speed records: Curve25519. See cr.yp.to/ecdh.html and papers.html#curve25519. Curve $y^2 = x^3 + 486662x^2 + x \mod 2^{255} - 19$; eliminate $y$; < 1000000 CPU cycles.


Many other public-key structures. e.g. “Public-key signing” reduces costs of securely sending a public message to many recipients.

Sender signs message independently of receiver, without any shared secrets.

Each receiver verifies signature; very fast computation.
Again improve cost constants, focus on useful $b$, etc.

Speed records: Curve25519. See cr.yp.to/ecdh.html and papers.html#curve25519. Curve $y^2 = x^3 + 486662x^2 + x \mod 2^{255} - 19$; eliminate $y$; < 1000000 CPU cycles.


Many other public-key structures. e.g. “Public-key signing” reduces costs of securely sending a public message to many recipients. Sender signs message independently of receiver, without any shared secrets. Each receiver verifies signature; very fast computation.
Again improve cost constants, focus on useful b, etc.

Speed records: Curve25519.
See cr.yp.to/ecdh.html and papers.html#curve25519.
Curve $y^2 = x^3 + 486662x^2 + x + 1$; eliminate $y$;
$< 1000000$ CPU cycles.

Many other public-key structures.
e.g. “Public-key signing” reduces costs of securely sending a public message to many recipients.
Sender signs message independently of receiver, without any shared secrets.
Each receiver verifies signature; very fast computation.

E.g. “Public-key encryption” generates new shared secret for each message.
Always increases costs, as far as I know.
The literature on “public-key encryption” is a historical accident.
Best to reuse one secret for many messages.
Minimize public-key operations.
Many other public-key structures.
e.g. “Public-key signing” reduces costs of securely sending a public message to many recipients.

Sender signs message independently of receiver, without any shared secrets.

Each receiver verifies signature; very fast computation.

e.g. “Public-key encryption” generates new shared secret for each message.

Always increases costs, as far as I know.

The literature on “public-key encryption” is a historical accident.

Best to reuse one secret for many messages.

Minimize public-key operations.
Many other public-key structures.

e.g. “Public-key signing”
reduces costs of securely
sending a public message
to many recipients.
Sender signs message
independently of receiver,
without any shared secrets.
Each receiver verifies signature;
very fast computation.

If large quantum computers
are built then they
will compute discrete logs quickly.
Huge effects on cryptography!
Some public-key systems seem to
survive quantum computers.
See PQCrypto 2006 abstracts:
postquantum.cr.yp.to

Exactly how fast are RSA,
DSA, ECDH, post-quantum
cryptosystems, etc.?
www.ecrypt.eu.org/ebats
is benchmarking public-key systems.
e.g. “Public-key encryption” generates new shared secret for each message.

Always increases costs, as far as I know.
The literature on “public-key encryption” is a historical accident.

Best to reuse one secret for many messages.
Minimize public-key operations.

If large quantum computers are built then they will compute discrete logs quickly. Huge effects on cryptography!

Some public-key systems seem to survive quantum computers. See PQCrypto 2006 abstracts: postquantum.cr.yp.to.

Exactly how fast are RSA, DSA, ECDH, post-quantum cryptosystems, etc.? www.ecrypt.eu.org/ebats is benchmarking public-key systems.